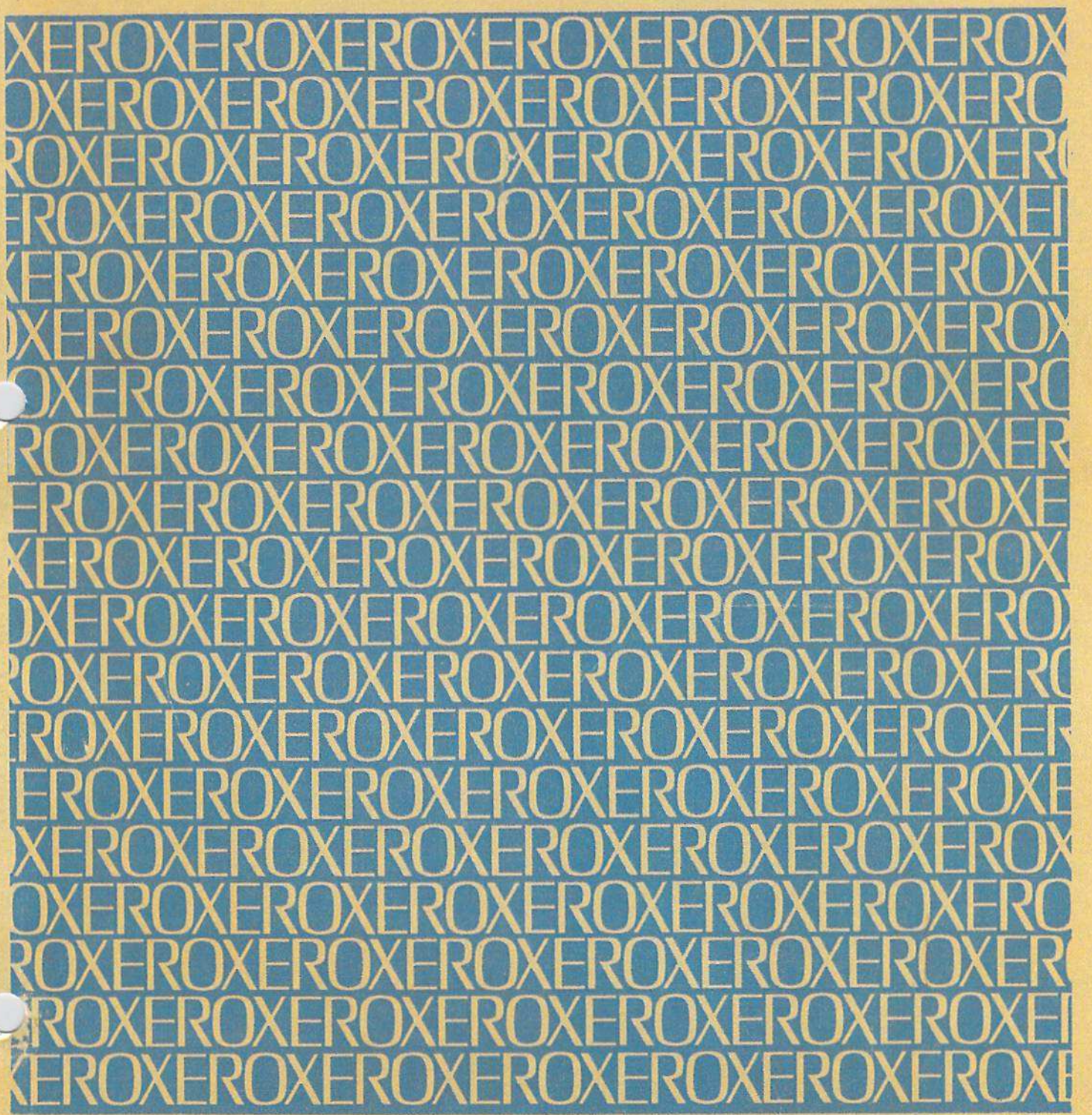
Xerox Extended FORTRAN IV

Sigma 5-9 Computers

Technical Manual



90 31 14A

Xerox Corporation
701 South Aviation Boulevard
El Segundo, California 90245
213 679-4511

XEROX

Xerox Extended FORTRAN IV

Sigma 5-9 Computers

Technical Manual

90 31 14A

June 1974

Price: $25.25

THIS PUBLICATION WAS PREPARED WITH TEXT AND PRINTED ON THE XEROX GRAPHICS PRINTER

© Xerox Corporation, 1974

Printed in U.S.A.

NOTICE

This manual documents the E00 version of the compiler.

TABLE OF CONTENTS

		PAGE
INTRODUCTION		14
Chapter 1	Overview	17
1.1	General Description of Xerox Sigma Extended FORTRAN IV	18
1.1.1	Externally	18
1.1.2	Internally	18
1.2	Overlay Structure	19
1.3	Brief Descriptions of the Compiler Assemblies	21
1.4	Lists	28
1.4.1	Dynamic Lists	28
1.4.2	How Lists are stored	30
1.4.3	Static Lists	33
1.4.4	List Pointers	34
1.4.5	Searching and Registering; Entry, key, and offset size	35
1.4.6	A few basic lists	36
1.4.7	Scratch Files	38
1.4.8	Initialize Lists (INILISTS)	39
1.4.9	Reassign Memory (REASSIGN)	41
1.4.10	Freight Trains	43
1.4.11	The List Procs	44
1.5	Table of Lists	46
1.6	POPs	92
1.6.1	True/False Feature	95
1.6.2	Special Addressing	97
1.6.3	The POP Procs	99
1.6.4	Naming Conventions	100

		PAGE
1.7	Tables of POPs	101
1.7.1	Explanation and Alphabetic Table	102
1.7.2	Functional Table of POPs (With Descriptions)	111
1.7.2.1	Manipulation	111
1.7.2.2	Arithmetic	115
1.7.2.3	Boolean	117
1.7.2.4	Memory Modification	119
1.7.2.5	Jump	121
1.7.2.6	Test	124
1.7.2.7	General List	129
1.7.2.8	Specific List	134
1.7.2.9	Pointer	140
1.7.2.10	Addressing	144
1.7.2.11	Plex	147
1.7.2.12	Print	149
1.7.2.13	Input Scan	152
1.7.2.14	Generating (Build)	154
1.7.2.15	Control	160
1.7.2.16	Debug	161
1.8	Recursion	162
1.8.1	POPs and Lists	162
1.8.2	Calling and Returning	162
1.8.3	Tricky Recureive Coding	163
1.9	Interpreter (INTERPTR)	165
1.9.1	How to Write and Use POPs	165

		PAGE
1.9.1.1	Assembly format	165
1.9.1.2	Special addresses	165
1.9.1.3	Word format	166
1.9.1.4	Effective address calculation	166
1.9.2	How to Write POPDs	167
1.9.2.1	Defining the POPD name	167
1.9.2.2	POPD/Interpreter interface	168
1.9.2.3	Register conventions in the Interpreter	170
1.9.2.4	Subroutines and special exits	171
1.9.2.5	Summary of POPD definition procedures	174
1.9.3	How to Write Interpretive Code	175
1.9.3.1	Machine/POP language interface	175
1.9.3.2	Subroutine linkage	175
1.9.3.3	Recursive exits	176
1.9.4	Interpreter/Debug Interface	178
1.10	Plexes	178
1.10.1	Representation of Expressions	181
1.10.1.1	Plexes used to represent arithmetic expressions	184
1.10.1.2	Plexes used to represent logical expressions	187
1.10.2	POPs Concerned with Plexes	190
1.10.2.1	Plex-building POPs	191
1.10.2.2	Plex-destroying POPs	192
1.10.3	Indexing by Plex Type	192
1.10.4	The Plex definition Procs	193
1.10.5	Other Uses	195
1.11	Pointers and Pseudo-pointers	197

		PAGE
1.12	The Spectrum of ID Field Numbers	198
1.13	Traits and Mode	204
1.13.1	Traits	204
1.13.2	Modes	209
1.13.3	Inheritance of traits	211
1.13.4	Inheritance of mode	211
1.13.5	Indexing by mode	213
1.14	Print POPs	213
1.15	Quotes	218
1.16	Terminology	219
1.17	Assembly Listing Conventions	221
1.17.1	Title cards	221
1.17.2	REFs and DEFs	221
1.17.3	Local Labels	222
1.17.4	Other Labels	222
1.17.5	Assembly columns	223
1.17.6	Comments	223
1.17.7	SYSTEM directives	225

		PAGE
Chapter 2	Specific Areas	226
2.1	Compiler Initialization	227
2.1.1	Initialization done only once	227
2.1.2	Initialization done before each program compiled	227
2.2	Character Scanning	228
2.2.1	Ready Scan Character (RDYSCNCH)	229
2.2.2	Next Active Character (NXTACTCH)	230
2.2.3	Next Input Character (NXTINPCH)	231
2.3	Card and Line Handling	232
2.3.1	Release Line (RELLINE)	234
2.3.2	Release Card (RELCARD)	235
2.3.3	Send Print Buffer (SENDPRBF)	236
2.3.4	Output Print Buffer (PBOUT)	237
2.3.5	Release Input (RELINPUTO	237
2.3.6	Print if Summary (PRIFSUM)	238
2.4	Character Translate Table (CHTRATBL)	239
2.5	Saving Alphanumeric Characters	241
2.6	Diagnostics	242
2.6.1	Format of Error Messages	242
2.6.2	Immediate Error Messages	243
2.6.3	Delayed Error Messages	244
2.7	TRY-Fail	246
2.7.1	Saving and Restoring Information	246
2.7.2	Overall Statement Scan and Fail Structure	248
2.7.3	Top-Down Programming	265
2.7.4	The TRY POP	270

		PAGE
2.8	Identifier Resolution	273
2.8.1	Permissibility	274
2.8.2	Symbol Table POPs	278
2.8.3	Type (or Mode) Resolution	280
2.8.3.1	Register Type (RETYPE)	280
2.8.3.2	Set Type (SETTYPE)	281
2.8.3.3	Impose Type if Possible (IMPTYPE)	281
2.8.3.4	Inherit Type if Possible (INHTYPE)	281
2.8.3.5	Set Element Size (SETELSIZ)	282
2.9	Overall Statement Scan Organization	282
2.9.1	Overall Line Control	282
2.9.2	Statement Scan (STASCAN)	285
2.9.3	Compound Statement Scan (COMSTASC)	286
2.9.4	Finish Up (FINISHUP)	287
2.9.5	Label Field Scan (LBLFLDSC)	289
2.9.6	Process Label (PROCLABL)	291
2.9.7	Process ends of DOs (PROCENDO)	292
2.9.8	File Code (FILECODE)	293
2.9.9	Statement Exits	295
2.9.9.1	Fail if BLOCK DATA (FAILIFBD)	295
2.9.9.2	Fail if Logical IF (FAILIFLI)	295
2.9.9.3	Fail if not end or if delayed fail (FAILIFNE)	295
2.9.8.4	Fail if in DO loop (FAILIFDO)	295
2.9.8.5	No Path Test (NOPATH)	295
2.9.8.6	Check Label after IF (CHKLAIF)	296

		PAGE
2.9.9.7	No label after IF (NOLBLAIF)	296
2.9.9.8	Non-FORMAT label (NONFORLB)	296
2.10	Miscellaneous Scan Routines	296
2.10.1	Comma Tests (COMMAT1, COMMAT2, COMMAT3, COMMAT4)	296
2.10.2	Excess right parenthesis scan (EXCSRPSC)	297
2.11	Pass 1/Pass 2 Interface	298
2.12	CODELIST Manipulation in Pass 1	306
2.13	CODELIST Manipulation in Pass 2	307
2.14	Code Generation	308
2.14.1	Statement Gen Loop (PROCSTAT)	308
2.14.2	Expression Generation	309
2.14.3	Address Generation	309
2.14.3.1	Address Calculation Gen (ADRCALC)	310
2.14.3.2	Guarantee Effective Address in Index (GTEAX)	315
2.14.3.3	Free/Busy an Index Register (FREEX, BUSYX)	316
2.14.4	Retroactive Subscript Gen (SCRPTGEN)	316
2.14.5	Delayed Code	321
2.15	Assemble Code (ASMCODE)	322
2.16	Naming Conventions for Expression Evaluation Generation Subroutines	323
2.17	Register Context Handling	328
2.17.1	X-reg Context Storage, XREGLIST	328
2.17.2	DO-reg Context Storage, DREGLIST	328
2.17.3	Remembering Register Context En Masse, TXREGLST, TDREGLST	331
2.17.4	Register Allocation	331
2.17.5	Register Clobbering, DO-reg Restoring	331
2.18	Register Usage in Object Code	332

		PAGE
2.18.1	X-regs	332
2.18.2	DO-regs	333
2.18.3	Accumulators	334
2.19	Temps	334
2.19.1	TEMPs, PTMPs, and RTMPs	334
2.19.2	Temp Pointers	335
2.19.3	Private Temps (PTMPs)	335
2.19.4	Ordinary Temps (TEMPs)	337
2.19.5	Privacy In Ordinary Temps	339
2.19.6	Which Comes First?	339
2.19.7	The Real-Time (RTMP)	340
2.20	Mode Conversion	340
2.20.1	The MODELIST	341
2.20.2	Mode Conversion Gen (MODCVGEN)	341
2.20.3	Constant Mode Conversion	342
2.21	Pass 2/Pass 3 Interface	342
2.22	Output Program (OUTPROG)	345
2.22.1	Some Typical Instruction Drivers	346
2.22.2	Some Rinky-Dink Drivers	348
2.22.3	Output Immediate Instruction (OUTIMMOP)	350
2.23	I/O Unit Refernece	350
2.24	Comments in Object Listing	351
2.25	Subprogram Calling and REceiving Sequences	352
2.25.1	General Considerations	352
2.25.1.1	Mathematical Functions: Drivers and Evaluators	353

		PAGE
2.25.1.2	Naming of System Routines	354
2.25.2	Standard Calling/Receiving Sequences	354
2.25.2.1	Standard Calling Sequences	354
2.25.2.1.1	Subprogram Names and Statement Labels as Arguments	357
2.25.2.1.2	Arguments That Are Themselves Dummies	358
2.25.2.2	Standard Receiving Sequences	360
2.25.2.3	Register Conventions in Standard Subprograms	364
2.25.2.4	Word Format in Calling and Receiving Sequences	364
2.25.2.5	Standard Accumulators for Arithmetic	366
2.25.2.6	Returning the Results of Functions	366
2.25.2.7	The Argument Set-Up Routines 9SETUPN, 9SETUP0, 9SETUP1, 9SETUP2	
2.25.2.8	Receiving Sequences Involving Multiple Dummies	367
2.25.2.9	The Argument Set-Up Routine 9SETUPV	368
2.25.2.10	The Argument Set-Up Routine 9SETUPM	369
2.25.3	Nonstandard Calling/Receiving Sequences	369
2.25.3.1	Mathematical Function Evaluators	369
2.25.3.2	Power Routines	371
2.25.3.3	Type Conversion Routines (also called Mode Conversion)	371
2.25.3.4	Complex Arithmetic Routines	372
2.26	Double-Word Shenanigans	373
2.26.1	Storage Allocation	373
2.26.2	Temps	375
2.26.3	Constants	375
2.27	Generate Literals (GENLIT)	375
2.28	Binary Output (OUTBO) (Pass 3)	378
2.28.1	Start Binary Output (STARTBO) (Pass 3)	380

PAGE

2.28.2 Output BO Datum (BODATUM) (Pass 3) 382

2.28.3 Finish Binary Output (FINISHBO) (Pass 3) 384

2.28.4 Object Language Expressions/Declarations 384

2.29 Object Listing 385

2.29.1 Prologue 387

2.29.2 Epilogue (FINISHLO) 387

2.29.3 Body (Interface items) 390

2.29.3.1 Not directly related to printing 390

2.29.3.2 Set up 390

2.29.3.3 Normal instructions 391

2.29.3. 3.1 Print Operand (PROPERND) 393

2.29.3.3.1.1 Absolute 393

2.29.3.3.1.2 Symbol Table 394

2.29.3.3.1.3 External reference to non-FORTRAN name 394

2.29.3.3.1.4 Label 394

2.29.3.3.1.5 Temp 394

2.29.3.3.1.6 Address Function 395

2.29.3.3.1.7 String constant 395

2.29.3.3.1.8 Print Constant Operand (PRINTCON) 395

2.29.3.4 Abnormal instructions 399

2.29.3.5 Other (not instructions) 400

2.30 Summary Listing (SUMMARY) 401

2.30.1 Initialize 401

2.30.2 Alphabetic Symbol Table Sort (ASTSORT) 402

2.30.3 Output Symbol Table (OST) 402

		PAGE
2.30.4	Output Label Table (OLT)	403
2.30.5	Output Functional Groups (OFG)	403
2.30.6	Output Final Statistics (OFS)	404
2.31	Abort Errors	405

		PAGE
Chapter 3	Throughviews	411
3.1	Identifiers	412
3.2	Arrays	415
3.3	Subscripts	419
3.4	Dummies	425
3.5	Multiple Dummies	427
3.6	Intrinsic Functions	428
3.6.1	Intrinsic Functions (Pass 1)	428
3.6.2	Intrinsic Function Generators (Pass 2)	430
3.7	Constants	431
3.7.1	Storage	431
3.7.2	Scanning	432
3.7.3	Generation	434
3.7.4	Listing	434
3.8	Allocation	434
3.8.1	Initialize Symbol Table (INISYMTB) (ALLOCATE)	435
3.8.2	Allocate Blank COMMON (ALBLNCOM) (ALLOCATE)	436
3.8.3	Allocate Labeled COMMON (ALLABCOM) (ALLOCATE)	438
3.8.4	Process EQUIVALENCE (PROCEQIV) (ALLOCATE)	438
3.8.4.1	Process EQUIVALENCE Group (PEQGROUP) (ALLOCATE)	438
3.8.4.2	Next Equiv Item (NXEQITEM) (ALLOCATE)	439
3.8.4.3	Left and Right	440
3.8.4.4	Process Equiv Pair (PEQPAIR) (ALLOCATE)	442
3.8.4.5	EQUIVALENCE The Pair (EQVPAIR) (ALLOCATE)	444
3.8.4.5.1	Swap Left And Right (SWAPLR) (ALLOCATE)	446
3.8.4.5.2	Build Clump From Left (BILCLMPL) (ALLOCATE)	446

		PAGE
3.8.4.5.3	Build Clump From Right (BILCLMPR) (ALLOCATE)	446
3.8.4.5.4	Handle Right Clump (HRCLUMP) (ALLOCATE)	447
3.8.4.5.5	Compute Right Displacements (COMPRDIS) (ALLOCATE)	
3.8.4.5.6	Check Parity Against Right Clump (CHECKPRTY)(ALLOCATE)	
3.8.4.5.7	Allocate Equiv Clump (ALOCLUMP) (ALLOCATE)	447
3.8.5	Allocate Local Variables (ALLOCALV) (ALLOCATE)	448
3.8.6	Allocate Dummies (ALDUMMYS) (ALLOCATE)	449
3.8.7	Check COMMON Parity (CKCOMPTY) (ALLOCATE)	449
3.8.7.1	The Parity Problem In COMMON Blocks	449
3.8.8	Finish Blank COMMON (FINBLCOM) (ALLOCATE)	450
3.8.9	Finish Labeled COMMON (FINLBCOM) (ALLOCATE)	450
3.8.10	Allocation Diagnostics Demonstration	451
3.9	Subprograms	452
3.9.1	Classification	452
3.9.2	Generation	453
3.10	Arithmetic Expressions	453
3.10.1	Arithmetic Expressions (Pass 1)	456
3.10.1.1	Entry Points	456
3.10.1.2	Hierarchy	459
3.10.1.2.1	Arithmetic Expression Scan (AREXPSC)	459
3.10.1.2.2	Signed Term Scan (SGNTRMSC)	459
3.10.1.2.3	Term Scan (TERMSCAN)	460
3.10.1.2.4	Factor Scan (FACTORSC)	460
3.10.1.2.5	Element Scan (ELEMSCAN)	461
3.10.1.3	Elements	463
3.10.1.3.1	Scalar Scan (SCALARSC)	463
3.10.1.3.2	Array Scan (ARRAYSCN)	463
3.10.1.3.3	Subscript Sequence Scan (SCRSEQSC)	464
3.10.1.3.4	Multiple Dummy Scan (MULDMYSC)	464
3.10.1.3.5	Function Scan (FUNCSCAN)	465

		PAGE
3.10.1.3.5.1	Argument Sequence Scan (ARGSEQSC)	465
3.10.1.3.5.2	Argument Scan (ARGSCAN)	465
3.10.1.3.5.2.1	Constant Argument Scan (CONSARSC)	466
3.10.1.3.5.2.2	Name Alone Argument Scan (NAMARGSC)	467
3.10.1.3.5.2.3	Statement Number Argument Scan (STNOARSC)	468
3.10.1.3.5.3	Process If Intrinsic (PRIFINTR)	468
3.10.1.3.6	Constant Scan (CONSTSCN)	469
3.10.1.3.7	Identifier Errors	470
3.10.2	Arithmetic Expression Generation (Pass 2)	472
3.10.2.1	PLEXGEN (Gen By Plex Type) (Pass 2)	473
3.10.2.2	SUMGEN and PRODGEN (Pass 2)	473
3.10.2.3	Quotient Gen (DIVGEN) (Pass 2)	475
3.10.2.4	Expon Gen (EXPONGEN) (Pass 2)	476
3.10.2.5	Mode Switch Gen (MODSWG) (Pass 2)	478
3.10.2.6	Function Call Gen (FCALLGEN) (Pass 2)	478
3.10.2.7	Replacement Gen (REPACGEN) (Pass 2)	478
3.10.2.7.1	Logical Replacement Gen (LOGREPG) (Pass 2)	479
3.10.2.7.2	Special Replacement Gen (SPECREPG) (Pass 2)	479
3.10.2.7.3	Ordinary Replacement Gen (ORDREPG) (Pass 2)	481
3.11	Logical Expressions	481
3.11.1	Logical Expressions (Pass 1)	482
3.11.1.1	Entry Points	

		PAGE
3.11.1.2	Hierarchy	483
3.11.1.2.1	Logical Expression Scan (LOGEXPSC)	484
3.11.1.2.2	OR-term Scan (ORTERMSC)	484
3.11.1.2.3	AND-term Scan (ANDTRMSC)	484
3.11.1.2.4	Logical Factor Scan (LOGFCTSC)	484
3.11.1.2.5	Logical Element Scan (LOGELMSC)	484
3.11.1.3	Relational Expression Scan (RELEXPSC)	485
3.11.1.3.1	Relational Operator and Arithmetic Expression (RELOPEXP)	486
3.11.1.3.2	Relational Operators (RELOPSGL, RELOPSEN)	486
3.11.2	Logical Expression Generation (Pass 2)	487
3.12	Real-Time	492
3.13	Control Card Options	494
3.14	Line Numbers	499
3.15	Plex Passing	501
3.16	Labels	502
3.17	Loops	503
3.17.1	Properly Nested Loops	504
3.17.2	Illegally Nested Loops	504
3.18	Source Lines	505
3.19	Run-Time Debug	505
3.19.1	Debug Interface Code	506
3.19.2	Debug Tables	507
3.20	ABNORMAL Statement (ABNORMST) (Pass 1 only)	510
3.21	ACCEPT TAPE Statement (ACCTAPST) (Pass 1 only)	510
3.22	ACCEPT Statement (ACCEPTST) (Pass 1 only)	510

		PAGE
3.23	ASSIGN Statement	511
3.23.1	ASSIGN Statement (ASSIGNST) (Pass 1)	511
3.23.2	ASSIGN Statement Gen (ASSIGNG) (Pass 2)	511
3.24	Assigned GO TO Statement	512
3.24.1	Assigned GO TO Statement (GOTOSTA) (Pass 1)	512
3.24.2	Assigned GO TO Statement Gen (ASNGOSTG) (Pass 2)	512
3.25	Assignment Statement	512
3.25.1	Assignment Statement (ASIGNMNT) (Pass 1)	513
3.25.2	Assignment Statement Gen (ASNSTAG) (Pass 2)	513
3.26	BACKSPACE Statement	513
3.26.1	BACKSPACE Statement (BKSPACST) (Pass 1)	514
3.26.2	BACKSPACE Statement Gen (BACKSPSG) (Pass 2)	514
3.27	BLOCK DATA Statement (BLKDATST) (Pass 1)	514
3.28	CALL Statement	514
3.28.1	CALL Statement (CALLSTA) (Pass 1)	515
3.28.2	CALL Statement 1 (CALLSTA1)	515
3.28.3	CALL Statement Gen (CALLSTG) (Pass 2)	515
3.28.4	Sprog Call Gen (SPCALLGN)	515
3.29	COMMON Statement (COMMONST) (Pass 1 only)	517
3.30	COMPLEX Statement (COMPLXST) (Pass 1 only)	517
3.31	Computed GO TO Statement	518
3.31.1	Computed GO TO Statement (GOTOSTA) (Pass 1)	518
3.31.2	Computed GO TO Statement Gen (COMPGOSG) (Pass 2)	518
3.32	CONNECT Statement	519
3.32.1	CONNECT Statement (CONECTST) (Pass 1)	519

		PAGE
3.32.2	CONNECT Statement Gen (CONECTSG) (Pass 2)	520
3.33	CONTINUE Statement (CONTINST) (Pass 1 only)	521
3.34	DATA Statement	522
3.34.1	DATA Statement (DATASTA) (Pass 1)	522
3.34.1.1	DATA Variable List (No DO) Scan (DVLNDSCN)	526
3.34.1.2	DATA Variable List (With DO) Scan (DVLWSCN)	527
3.34.1.3	DATA Variable Item Scan (DVARSCAN)	528
3.34.1.4	DATA Expression Scan (DEXPSCAN)	528
3.34.1.5	Check DATA Subscript Variables (CHKDSUBS)	529
3.34.1.6	DATA Constant List Scan (DCONLSCN)	529
3.34.2	Expand DATA Pair Sequence (EXPDPRSQ)	530
3.34.2.1	Process DATA Variable Item (PDVITEM)	532
3.34.2.2	Process DATA Constant Item (PDCONITM)	537
3.34.3	DATA Statement (Passes 2 and 3)	537
3.34.3.1	Begin DATA Gen (BDATAGEN) (Pass 2)	537
3.34.3.2	End DATA Gen (EDATAGEN) (Pass 2)	537
3.34.3.3	DATA Pair Gen (DPAIRGEN) (Pass 2)	538
3.34.3.4	DATA Item Gen (DTITEMGN) (Pass 2)	538
3.34.3.5	DATA Statement in Pass 3	539

		PAGE
3.35	DECODE Statement	539
3.35.1	DECODE Statement (DECODEST) (Pass 1)	539
3.35.2	DECODE Statement Gen (DECODESG) (Pass 2)	540
3.36	DIMENSION Statement (DIMENSTA) (Pass 1 only)	540
3.37	Allow Dimensions (ALLOWDIM)	540
3.38	DO Statement	545
3.38.1	DO Statement (DOSTA) (Pass 1)	545
3.38.1.1	DO-Control (DOCONTRL)	546
3.38.1.2	DO Statement Number Scan (DOSTANUM)	546
3.38.1.3	Outer DO Check (OUTERDO)	546
3.38.2	DO Statement Gen (DOSTATG) (Pass 2)	546
3.38.3	End of Loop Gen (ENDLOOPG) (Pass 2)	548
3.39	DOUBLE COMPLEX Statement (DBLPRCST) (Pass 1 only)	549
3.40	DOUBLE PRECISION Statement (DBLPRCST) (Pass 1 only)	549
3.41	ENCODE Statement	549
3.41.1	ENCODE Statement (ENCODEST) (Pass 1)	550
3.41.2	Encode-Decode (ENCODECO)	550
3.41.3	ENCODE and DECODE Statement Gen (ENCODESG and DECODESG) (Pass 2)	550
3.42	END Statement	553
3.42.1	END Statement (ENDSTA) (Pass 1)	553
3.42.2	END Statement Gen (ENDSTATG) (Pass 2)	553
3.43	END FILE Statement	554
3.43.1	END FILE Statement (ENDFILST) (Pass 1)	554
3.43.2	ENDFILE Statement Gen (ENDFILSG) (Pass 2)	554
3.44	END LABELS Statement (ENDLABST) (Pass 1 only)	554

		PAGE
3.45	END Labels Subroutine (ENDLBLS)	555
3.46	End of Logical IF (Pass 2)	555
3.47	ENTRY Statement	555
3.47.1	ENTRY Statement (ENTRYSTA) (Pass 1)	555
3.47.2	ENTRY Statement Gen (ENTRYSTG) (Pass 2)	557
3.48	EQUIVALENCE Statement (EQUIVSTA) (Pass 1 only)	558
3.49	EXTERNAL Statement (EXTERNST) (Pass 1 only)	560
3.50	FORMAT Statement	560
3.50.1	FORMAT Statement (FORMATST) (Pass 1)	560
3.50.1.1	FORMAT List Scan (FORLSTSC)	561
3.50.1.2	FORMAT Element Scan (FORMELSC)	561
3.50.1.3	Next Hollerith Character (NXTHOLCH)	562
3.50.2	FORMAT Statement Gen (FORMATSG) (Pass 2)	562
3.50.3	FORMAT Scan at Run-Time (9IEDIT)	563
3.51	FREQUENCY Statement (FREQSTA) (Pass 1 only)	564
3.52	FUNCTION Statement	564
3.52.1	FUNCTION Statement (FUNCTSTA) (Pass 1)	564
3.52.1.1	FUNCTION Common 3 (FUNCOM3)	566
3.52.1.2	Multiple Dummy Scan (MDMYSCN)	569
3.52.2	FUNCTION Statement Gen (FUNCSTAG) (Pass 2)	569
3.53	GLOBAL Statement (GLOBALST) (Pass 1 only)	569
3.54	GO TO Statement	570
3.54.1	GO TO Statement (GOTOSTA) (Pass 1)	570
3.54.2	Ordinary GO TO Statement Gen (GOTOSTAT) (Pass 2)	570
3.55	IF Statements	570
3.55.1	IF Statements (IFSTAS) (Pass 1)	570

PAGE

3.55.1.1 Arithmetic IF Statement (Pass 1) 571

3.55.1.2 IF SENSE LIGHT Statement (Pass 1) 572

3.55.1.3 IF SENSE SWITCH Statement (Pass 1) 572

3.55.1.4 IF FLOATING OVERFLOW Statement (Pass 1) 574

3.55.1.5 Logical IF Statement (LOGIFSTA) (Pass 1) 574

3.55.2 IF Statements (Pass 2) 576

3.55.2.1 Arithmetic IF Statement Gen (AIFSTAG) (Pass 2) 576

3.55.2.2 IF SENSE LIGHT Statement Gen (IFSLSTG) (Pass 2) 577

3.55.2.3 IF SENSE SWITCH Statement Gen (IFSWSTG) (Pass 2) 578

3.55.2.4 IF OVERFLOW Statement Gen (IFOVFLSG) (Pass 2) 578

3.55.2.5 Logical IF Statement Gen (LOGIFSTG) (Pass 2) 578

3.56 IMPLICIT Statement (IMPLCTST) (Pass 1 only) 581

3.57 In-line Assembly Code 582

3.57.1 In-line Assembly Code (SINCOLM1) (Pass 1) 582

3.57.1.1 Op Code List 583

3.57.1.2 Scanning In-line Assembly Code 584

3.57.1.3 Address Scan Subroutines 587

3.57.1.3.1 Scan Value (SCANVAL) 587

3.57.1.3.2 Scan Address (SCANADR) 587

3.57.1.3.3 Scan Special Address (SCANSPAD) 589

3.57.2 In-line Machine Code Statement Gen (CODESTAG) (Pass 2) 589

3.58 INPUT Statement 590

3.58.1 INPUT Statement (INPUTSTA) (Pass 1) 590

3.58.2 INPUT with List Gen (INPUTLSG) (Pass 2) 590

3.58.3 INPUT Without List Gen (INPUTSTG) (Pass 2) 590

		PAGE
3.59	INTEGER Statement (INTGERST) (Pass 1 only)	591
3.60	Input/Output List	591
3.60.1	I/O List Scan (IOLIST) (Pass 1)	591
3.60.1.1	End of I/O List (ENDIOL) (Pass 1)	592
3.60.1.2	I/O DO-Control (IODOCTRL) (Pass 1)	592
3.60.1.3	I/O Item (IOITEM) (Pass 1)	593
3.60.1.4	I/O List Scan within parentheses (IOLISTSC)	595
3.60.2	I/O List Generation (Pass 2)	596
3.60.2.1	I/O Item Gen (IOTMGEN) (Pass 2)	596
3.60.2.1.1	Theory	596
3.60.2.1.2	Details	599
3.60.2.2	I/O DO-Control Gen (IODOGEN) (Pass 2)	602
3.60.2.3	End of I/O List Gen (ENDIOLG) (Pass 2)	602
3.61	LOCAL FUNCTION, SUBROUTINE, or ENTRY Statement (LOCALSTA) (Pass 1 only)	
3.62	LOGICAL Statement (LOGICSTA) (Pass 1 only)	603
3.63	NAMELIST Statement	603
3.63.1	NAMELIST Statement (NAMELSTA) (Pass 1)	603
3.63.2.1	List Name on Name List (LISTNAME)	603
	NAME LIST Gen (NAMELGEN) (Pass 2)	604
3.64	Null Statement (two kinds)	606
3.64.1	Null Statement (Pass 1)	606
3.64.2	Null Statement (Pass 2)	607
3.65	OUTPUT Statement (OUTPUTST) (Pass 1 only)	608
3.65.1	Initialization	610
3.65.2	Interface with I/O List Scan	611
3.65.2.1	Open OUTPUT Item (OPENOUT)	613
3.65.2.2	Close OUTPUT Item (CLOSEOUT)	613
3.65.2.3	Finish OUTPUT Group (FINOUT)	613
3.65.2.4	FORMAT Spec by Mode (FSBYMODE)	613
3.65.2.5	Load Alpha (LODALPHA)	613

		PAGE
3.66	PAUSE Statement	615
3.66.1	PAUSE Statement (PAUSESTA) (Pass 1)	615
3.66.2	Integer or Zero (INTORZ)	615
3.66.3	PAUSE Statement Gen (PAUSESTG) (Pass 2)	616
3.67	PRINT Statement (PRINTSTA) (Pass 1 only)	616
3.68	PUNCH TAPE Statement (PCHTAPST) (Pass 1 only)	616
3.69	PUNCH Statement (PUNCHSTA) (Pass 1 only)	616
3.70	READ Statements	617
3.70.1	READ Statements (READSTAS) (Pass 1)	617
3.70.1.1	READ-WRITE Scan (RDWRSCAN)	617
3.70.1.2	FORTRAN II READ Statements (F2RDSTAS) (Pass 1 only)	618
3.70.1.2.1	READ INPUT TAPE Statement (Pass 1 only)	619
3.70.1.2.2	READ TAPE Statement (Pass 1 only)	619
3.70.1.2.3	READ DRUM Statement (Pass 1)	619
3.70.1.2.4	READ DISC Statement (Pass 1)	619
3.70.1.2.5	READ Cards Statement (Pass 1 only)	620
3.70.1.2.6	FORTRAN II I/O Scan (F2IOSCAN)	620
3.70.1.2.7	Drum Scan and Disc Scan (DRUMSCAN and DISCSCAN)	621
3.70.1.3	FORMAT Reference Scan (FORMREF)	621
3.70.1.4	Array Name Scan (ARYNAMSC)	622
3.70.2	BCD READ Statment Gen (BCDREADG) (Pass 2)	622
3.70.3	Binary READ Statement Gen (BINREADG) (Pass 2)	623
3.70.4	BCD READ with END=/ERR= Statement Gen (BCDRDEEG) (Pass 2)	623
3.70.5	Binary READ with END=/ERR= Statement Gen (BINRDEEG) (Pass 2)	624
3.70.6	READ DRUM Statement Gen (RDDRUMSG) (Pass 2)	624
3.70.7	READ/WRITE DRUM/DISC (RWDRMDSC) (Pass 2)	624
3.70.8	READ DISC Statement Gen (RDDISKSG) (Pass 2)	624
3.71	READ Statement (REALSTA) (Pass 1 only)	625
3.72	REPEAT Statements	625
3.72.1	REPEAT Statements (REPEATST) (Pass 1)	625
3.72.1.1	REPEAT WHILE Statement (Pass 1)	625
3.72.1.2	REPEAT FOR Statement (Pass 1)	626

		PAGE
3.72.2	REPEAT WHILE Statement Gen (REPWHSTG) (Pass 2)	627
3.72.3	REPEAT FOR Statement Gen (REPEATSG) (Pass 2)	628
3.73	RETURN Statement	632
3.73.1	RETURN Statement (RETURNST) (Pass 1)	633
3.73.2	RETURN Statement Gen (RETURNSG) (Pass 2)	633
3.73.3	Asinine RETURN Statement Gen (ASRETSTG) (Pass 2)	634
3.73.4	RETURN Statement at Run-Time (9ASININE)	635
3.74	REWIND Statement	636
3.74.1	REWIND Statement (REWINDST) (Pass 1)	636
3.74.2	REWIND Statement Gen (REWINDSG) (Pass 2)	636
3.75	SENSE LIGHT Statement	637
3.75.1	SENSE LIGHT Statement (SNSLITST) (Pass 1)	637
3.75.2	SENSE LIGHT Statement Gen (SLITESG) (Pass 2)	637
3.76	Statement Function	637
3.76.1	Statement Function Definition (ASFD) (Pass 1)	637
3.76.1.1	Create New ID Entry (CREATEID)	640
3.76.1.2	Dekludge (DEKLUDGE)	640
3.76.2	Statement Function Gen (ASFDSTAG) (Pass 2)	642
3.77	STOP Statement	642
3.77.1	STOP Statement (STOPSTA) (Pass 1)	642
3.77.2	STOP Statement Gen (STOPSTAG) (Pass 2)	642
3.78	SUBROUTINE Statement	643
3.78.1	SUBROUTINE Statement (SUBRSTA) (Pass 1)	643
3.78.2	SUBROUTINE Statement Gen (SUBSTATG) (Pass 2)	643
3.78.3	Sprog Head Gen (SPHEADGN) (Pass 2)	645
3.78.3.1	Special Sprog Setup Code Gen (RTSPHDGN, DBSPHDGN) (Pass 2)	646
3.78.3.2	Receiving Sequence Gen (RSEQGEN) (Pass 2)	647
3.78.3.3	Generate Array Offsets (GENOFFST) (Pass 2)	647
3.79	TYPE Statement (TYPESTA) (Pass 1 only)	649
3.80	Type Statements (TYPESTAS) (Pass 1 only)	649
3.80.1	Secondary Asterisk Scan Table (ASTSCAN)	651

		PAGE
3.80.2	Type Functions (LOCTYCOM)	651
3.81	WRITE Statements	651
3.81.1	WRITE Statements (WRITSTAS) (Pass 1)	651
3.81.1.1	FORTRAN II WRITE Statements (F2WRSTAS) (Pass 1 only)	652
3.81.1.2	WRITE OUTPUT TAPE Statement (Pass 1 only)	652
3.81.1.3	WRITE TAPE Statement (Pass 1 only)	652
3.81.1.4	WRITE DRUM Statement (Pass 1)	652
3.81.1.5	WRITE DISC Statement (Pass 1)	652
3.81.2	BCD WRITE Statement Gen (BCDWRITG) (Pass 2)	653
3.81.3	Binary WRITE Statement Gen (BINWRITG) (Pass 2)	653
3.81.4	WRITE DRUM Statement Gen (WRDRUMSG) (Pass 2)	653
3.81.5	WRITE DISC Statement Gen (WRDISKSG) (Pass 2)	653

		PAGE
Chapter 4	External Mechanics	655
4.1	Theory of Monitor Interface	656
4.2	Input/Output Operations	657
4.2.1	Read a source line from SI. (SIREAD)	657
4.2.1.1	READ	658
4.2.1.2	DOREAD	659
4.2.2	Write a line on LO. (LOWRITE)	661
4.2.2.1	WRITE	661
4.2.2.2	DOWRITE	662
4.2.3	Write a record on GO. (GOWRITE)	662
4.2.4	Write a record on BO. (BOWRITE)	662
4.2.5	Write a record on SO. (SOWRITE)	663
4.2.6	Write a line on DO. (PRINTDOML)	663
4.2.7	Write a record on a scratch file. (XWRITE)	664
4.2.8	Read a record from a scratch file. (XREAD)	665
4.2.9	Rewind scratch file. (REWIND, REWINDML)	665
4.2.10	Top of page on LO. (LOPAGE)	666
4.2.11	Read a debug line from DI. (DIREAD)	667
4.2.12	Read one character from OC. (OCREADCH)	667
4.2.13	Write a line on OC. (OCWRITE)	667
4.3	Miscellaneous Operations	668
4.3.1	Get list memory (i.e. working storage). (GETLMEM)	668
4.3.2	Get additional list memory. (GETAMEM)	669
4.3.3	Interpret control card options. (FOPTIONS)	669
4.3.4	Load overlay segment. (LPASS1ML, LPASS2ML, LPASS3ML)	672
4.3.5	Exit to Monitor. (BPMEXIT)	673
4.3.6	Abort to Monitor. (ABORTML,BPMABORT, ILPRABRT)	673
4.3.7	Determine number of lines remaining on LO page. (NPRLINES)	674
4.3.8	Test sense switch. (SENSWICH)	674

		PAGE
4.4	Special Operations	674
4.4.1	Open files. (INITIAL)	674
4.4.2	Close files. (BPMEXIT, ABORTML)	676
4.4.3	Set up traps. (INITIAL)	677
4.4.4	Test for correspondence of files.	677
4.4.4.1	Test if SI equals LO. (SIEQLOML)	677
4.4.4.2	Test if LO equals specified file. (LOEQFILE)	677
4.4.4.3	Test if LO equals typewriter. (LOEQTYML)	678
4.4.4.4	Test for SI from typewriter. (TYINPUT, SIONLINE)	678
4.4.4.5	Test if specified file is assigned to a typewriter. (TYTEST)	678
4.4.5	Enter Master mode. (MASTER)	678
4.4.6	Enter Slave mode. (SLAVE)	678
4.5	Other Considerations	679
4.5.1	Entrance from Monitor	679
4.5.2	Restart	679
4.5.3	Consecutive compilations	679
4.5.4	Paging conventions	681
4.5.5	Error severity	684
4.5.6	64K restriction	684
4.5.7	Instruction sets and assembly language	685
4.5.8	Control sections and protection types	686
4.5.9	Program size, REF's and DEF's	686
4.5.10	Run-time I/O unit numbers and DCB's	688
4.5.11	Direct run-time Monitor calls	689
4.5.12	Hardware stacks	689
4.5.13	Arithmetic overflow and other traps	690
4.5.14	COMMON storage	691
4.5.15	Blank COMMON	692
4.5.16	Special provisions for the DATA and NAME LIST statements	692
4.5.17	Top of page awareness	693
4.5.18	Overlay structure details	695

TABLES

		PAGE
Table 1.7.1	Alphabetic Table of POPs	102
Table 1.9.1	Recursive Exits	177
Table 1.10.1	Arithmetic Plexes	185
Table 1.10.2	Logical Plexes	188
Table 1.10.3	Logical Plexes (Pass 2 only)	189
Table 1.11.1	Pointers and Such	197
Table 1.12.1	The Spectrum of ID Field Numbers	199
Table 2.8.1	Identifier Permissibility	276
Table 2.11.1	Structure of Pass 1/Pass 2 Interface Items	302
Table 2.14.1	Address Calculation Generation	312
Table 3.2.1	Symbol Table Structure of Array Names	417

Introduction

This manual documents the Xerox Sigma 5/9 Extended FORTRAN IV compiler. It is intended primarily for the systems programmer who needs to maintain, modify, or debug the compiler itself. It assumes some prior familiarity with the Sigma computer and considerable familiarity with the Extended FORTRAN IV language and operating characteristics. We recommend reading the Xerox Extended FORTRAN IV/LN Reference Manual (90 09 56) and the Xerox Extended FORTRAN IV/OPS Reference Manual (90 11 43) first. In Chapter 4, some familiarity with the RBM, BPM, and UTS operating systems is assumed.

It is not intended as a "text book" introduction to compiler design, although it contains a fair amount of our philosophy about good compiler design. It specifically documents this compiler, and should be used in conjunction with a complete listing of the compiler. Although some of the more general discussions (such as in Chapter 1) are meaningful by themselves, much of the rest is meant to be used in addition to the listings and the comments thereon. We have not in general bothered to take comments out of the listings and reproduce them in this manual. The theory was to go into greater detail explaining what was going on (in an overall sense) and why.

these are the areas not covered by this manual: 1) the compile-time debug package (F4DEB). 2) the run-time library.

The manual is divided into four chapters:

Chapter 1. Overview. An overview of the compiler, highlighting techniques and routines that are widely used throughout the compiler.

Chapter 2. Specific Areas. Specific areas that are genera ly within one pass or another, but are associated with no particular statements.

Chapter 3. Throughviews. All of the statements and a variety of language constructs are followed through the compiler from Pass 1 to Pass 2 and occasionally into Pass 3 and the run-time.

Chapter 4. External Mechanics. External mechanics that have nothing in particular to do with compiling the FORTRAN language. This is primarily monitor interface.

Chapter 1 is probably a good place to start, in order to get familiar with the terminology and the use of POPs and lists and plexes. The general terminology defined in Section 1.16 is applicable to the entire document. The other chapters can be referenced as needed, but anyone working heavily with the compiler should read the whole document through at least once. A general introduction to the compiler might be obtained by reading the following sections (here a section number, such as 2.7, does not include its subsections, such as 2.7.1, unless specifically mentioned, or unless the word "all" appears).

1.1 (all)	General Description
1.2	Overlay Structure
1.3	Brief Descriptions of the Compiler Assemblies
1.4.1	Dynamic Lists
1.4.2	How Lists are stored
1.4.6	A few basic lists
1.6	POPs
1.6.1	True/False Feature
1.6.2	Special Addressing
1.8 (all)	Recursion
1.10	Plexes
1.10.1 (all)	Representation of Expressions
1.11	Pointers and Pseudo-pointers
1.13	Traits and Mode
1.13.2	Modes
1.13.3	Inheritance of traits
1.13.4	Inheritance of mode
1.16	Terminology
2.6 (all)	Diagnostics
2.7	TRY-Fail
2.7.2	Overall Statement Scan and Fail Structure
2.7.3	Top-Down Programming
2.8	Identifier Resolution
2.8.1	Permissibility
2.9	Overall Statement Scan Organization
2.9.1	Overall Line Control

2.9.2 Statement Scan (STASCAN)

2.9.9 Statement Exits

2.11 Pass 1/Pass 2 Interface

2.29 Object Listing

2.31 Abort Errors

3.1 Identifiers

3.7 (all) Constants

3.10 Arithmetic Expressions

3.11 Logical Expressions

3.11.1 Logical Expressions (Pass 1)

4.1 Theory of Monitor Interface

In addition, we feel it is highly desirable, when Xerox considers designing new monitors and/or interface environments, that those involved take into account the areas covered in Chapter 4. There are a surprising number of problems in compiler/monitor interface, and they should preferably be resolved in advance. We feel that this chapter is a good starting place for collecting future interface considerations. A similar collection should of course be made for run-time interface considerations.

When a section title refers specifically to a section of coding in the compiler (as opposed to more general comments), the assembly label used for the coding is also shown in parentheses (e.g. see Section 2.2.1).

One of the drawbacks to a document as detailed as this is the problem of keeping it up to date. Almost any change to the compiler will necessitate corresponding changes in this manual, if it is to accurately reflect what the compiler really does. Currently the sections and subsections are arranged in a meaningful order. If new information needs to be added, or old information removed, the problem of renumbering to maintain optimum ordering will be immense, and we recommend that it not be done,especially because of the extensive cross-referencing by section number. We feel that the lesser of two evils is to leave gaps where necessary, and to add sections at the end of a chapter, even though they rightly belong somewhere else.

CHAPTER 1. OVERVIEW

An overview of the compiler, highlighting techniques and routines that are widely used throughout the compiler.

1.1 General Description of Xerox Sigma Extended FORTRAN IV

1.1.1 Externally

Xerox Extended FORTRAN IV contains a large number of language features not found in other FORTRAN systems. Some are relaxations of typical restrictions. Some are rather complicated features that provide interesting capabilities to experienced programmers. Others are specifically designed to make FORTRAN easier to use. (See publication #64-72-23A, "Making FORTRAN Easier to Use.") Also included are features for real-time operation, including re-entrant object code and in-line assembly language.

Considerable emphasis is placed upon clear and precise diagnostics. The compiler will go to some trouble in order to detect as many errors as possible, to report the cause and location of the errors, and whenever possible to recover from them. Similarly, the object code listing and summary were designed from the standpoint of, "What might the user want to know?" and therefore they strive to be as readable and complete as possible. (See NSS Memos #69-20-22, 6#69-20-33, and #69-78-02.)

The compiler does primarily local optimization of the generated object code. That is, the code for any given statement, considered by itself, is often extremely efficient. The global optimization (across groups of statements, such as DO loops) is rather limited, since no additional passes are made to work on this. The compiler will attempt to maintain subscripts and control variables in index registers within DO loops, but does not perform many optimizations of this kind that would be made possible by knowing in advance what is going on in the loops.

The language was designed to be upward compatible with ANS FORTRAN IV (i.e. it is a superset of ANS FORTRAN IV). It is also for the most part, compatible with FORTRAN II. (This is accomplished primarily by allowing the FORTRAN II input/output statements.) It also contains a variety of features that are needed only for compatibility with other FORTRAN systems; the facilities those systems provide can be accomplished more simply using other features in Xerox Extended FORTRAN IV. Because Sigma is a word-oriented machine, provisions for half-word and byte quantities as available in the IBM 360 H-level FORTRAN have not been implemented.

1.1.2 Internally

The compiler has three passes; that is, it passes over the source program (in one form or another) three times. It requires at least 16K words of storage, which includes itself and working storage. It also requires 2 scratch files, though any given compilation may not use them if working storage is sufficient. It is written in Meta-Symbol and uses only the Sigma 5 subset of op-codes. It uses no floating point, decimal, or privileged instructions. (In other words, it assembles with SYSTEM SIG5.) The largest assembly (PASS2F) is about 9400 lines.

Most of the compiler is written in an interpretive language called POPs (Programmed Operators -- this name derives from a hardware feature on the 900-Series computers). The POPs provide an elaborate instruction set that includes list manipulation, recursive calls, input scanning, code generation, diagnostic creation, printing, etc. The listings of many parts of the compiler thus look quite unlike Sigma coding. In fact, they look more like the 9300 FORTRAN IV compiler coding, from which large sections were taken almost verbatim. The interpreter for these POPs is of course written in machine language, as are various other subroutines. It is straightforward to transfer back and forth between POP language and machine language.

The tables created by the compiler are called lists, and they are dynamically allocated. That is, they have no fixed locations or sizes; they may slide around and expand and contract as needed. The POPs automatically perform memory reassignment when one list bumps into another. Although this may sound time-consuming, it generally occupies only a small percentage of compile time, and simplifies many other things.

One of those things is recursive coding. Most POP coding can, if desired, call itself. For example, one element of an expression can be another expression within parentheses, so Expression Scan calls itself. Even Statement Scan calls itself following a Logical IF. This is facilitated because most information is maintained on lists, including the return addresses for subroutines. These are kept on the EXITLIST, and the JRS POP (Jump Recursively to Subroutine) automatically handles the recursion.

Most of the constructs within FORTRAN statements are represented internally in a tree structure known as a plex, as opposed to other forms such as Polish notation.

To simplify the operation of the compiler under various Monitors, all of the interface with the outside world is concentrated in one assembly. This assembly (called MERRIMAC) provides simple calling sequences to the rest of the compiler for things such as input/output, and it is generally the only assembly that needs to be changed to interface with a different operating system.

1.2 Overlay Structure

The compiler is segmented into a root and three overlays. The common (unofficial) names for these are Pass 0, Pass 1, Pass 2, and Pass 3. The official names are PASS0F, PASS1F, PASS2F, and PASS3F. These are the names by which the Monitor's segment loader knows the four segments. This is explained more fully in "Overlay Structure Details", Section 4.5.18.

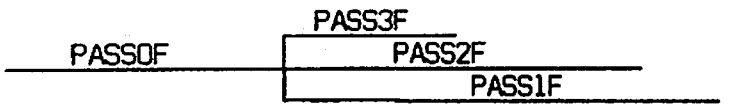
Pass 0 contains the sections needed by all three of the other passes. This includes: overall control of the compiler; the interpreter; most of the POPs; all input/output and other Monitor interface; miscellaneous subroutines; definitions of lists, plexes, traits, and other values; and temporary (or context) storage for all the passes.

Pass 1 reads the source program and outputs a digested version of it suitable for code generation by Pass 2. It also produces the source listing and most of the diagnostics, and builds tables (such as the Symbol Table) for use by later passes.

Pass 2 does storage allocation (determined by COMMON and EQUIVALENCE statements) and then reads the output from Pass 1, generates code for it, and performs a first assembly pass on that code.

Pass 3 reads the output from Pass 2 and performs the final assembly pass on it. It allocates temps and literals, and it generates the binary object program, the object listing, and the summary (as requested).

The overlay structure is shown below, along with the assemblies that fall into each segment. It should be noted, however, that some of these assemblies may not be included under certain Monitors.



PASS0F	PASS1F	PASS2F	PASS3F
PASS0F	PASS1F	PASS2F	PASS3F
MERRIMAC	STASCAN	ALLOCATE	GENLIT
FORTRAN4	EXSCAN	PATCH2	OUTPROG
INTERPTR	CONSCAN		OUTBO
LISTDEFS	PASS1ML		PRINTCON
FORTTEMP	SINCOLM1		SUMMARY
X1X2DCBS	PATCH1		PATCH3
P2P3ROOT			
REF			
PATCH			
PATCHT			

The diagram above reflects approximately the relative sizes of the segments. The path PASS0F-PASS1F is by far the largest and thus determines the memory requirements of the compiler. When the compiler is initially brought into memory, only PASS0F comes in. The assembly FORTRAN4 decides when to load the various overlays, and calls routines in MERRIMAC to perform this.

1.3 Brief Descriptions of the Compiler Assemblies

The previous section showed the assemblies that were located in each overlay segment. This section gives the highlights of each assembly (what its purpose is). The names given bleow are the names of the files in which each assembly's ROM is kept. Usually this is the same as the name by which the assembly is known within the compiler itself. (See Overlay Structure Details, 4.5.18.) Where it differs, the internal name is shown in parentheses.

Most of the compiler modules are assembled under SYSTEM FORTCOMP. This compressed source mdoule contains definitions of assembly Procedures (Procs) and values needed by most of the compiler -- particularly, those areas written in POPs. Though this module by itself produces no object module, and therefore does not appear in the overlay structure diagram, it does belong to the collection of compiler assemblies and is described below.

PASSOF PASSO

Contains subroutines that are useful to more than one pass. These include subscript manipulation, constant conversions, constant registering, scratch file manipulation, error printing, and some POPs that recurse (i.e. contain POPs within themselves). Also included are a few miscellaneous values used by more than one pass. Some of these probably should be in LISTDEFS.

MERRIMAC

Provides all of the interaction with the Monitor. The compiler is designed such that, for the most part, the other assemblies are independent of the characteristics of the operating system. (This is not entirely true. See Chapter 4, which documents Monitor interface.) When any assembly needs a hardware service, it calls a routine in MERRIMAC, with a simple calling sequence.

Among the services provided are: all input/output operations; requesting working storage from the Monitor; loading the compiler's overlays; testing the pseudo sense switches (used only by the compiler debug package); and interpreting the options on the !FORTRAN control card.

FORTRAN4

This contains the overall control of the compiler. It is the "main program", so to speak, because it is the only assembly with a starting address specified on the END card. It causes the various passes to be loaded and executed at the proper times. (Each pass is a subroutine and is called with a JRS POP.) It also determines the movements of the scratch tapes (or files) and causes control cards to be read and interpreted (by MERRIMAC). The compiler is initiated by branching to FORTRAN4 in machine language (i.e. with a B instruction, not a BRU POP). Depending on various options, the compiler may do one or several compilations. Then it exits by branching (in machine language) to BPMEXIT in MERRIMAC. BPMEXIT closes and saves various files,

then returns to the Monitor with an M:EXIT CAL. (In abort cases, it uses an M:XXX CAL.) Although entry and exit for FORTRAN4 are in machine language, most of the assembly is coded in POPs.

INTERPTR this is the interpreter for the FORTRAN POP language. It contains the interpreter itself, most of the POPs, Reassign Memory (which shufffles the lists around), and a few subroutines that are used primarily by the POPs but are occasionally called elsewhere. Note that some of the POPs are used only by a particular pass (e.g. ERC,BAF) and are therefore not in the resident (Pass 0) but in the appropriate pass. In that case, however, there is a branch in INTERPTR that leads to the POPD (POP Definition). (This is because the POP location field is only 12 bits, which means that the POP starting address must be within 4K of the POPBASE in INTERPTR.) INTERPTR also contains all of the POP and machine language "exits" used in POP coding (e.g. EXIT, EXITFAML, etc.).

LISTDEFS

This is the main control of the parameterization, or data structure, of the compiler. It contains the definitions of all the lists, plexes, Pass 2/3 operators, Symbol Table class and type values, traits and permissibility bits, and various masks. There are a small number of static lists that are physically located within some other assembly (e.g. the INTRILST, which names the intrinsic functions recognized by Pass 1), but these are also defined in LISTDEFS; the only difference is that Reassign Memory is not allowed to know about these. LISTDEFS contains no executable code, only constants and list-parameter temps.

FORTTEMP

Contains all the compiler's temps, both local and global. This includes the Character Translate Table, which is now a temp area because of the IMPLICIT statement. Global temps are externally DEFed. Local temps are accessed by each assembly by use of the TEMP proc, which references an area set aside in FORTTEMP. There is a unique area set aside for each assembly, and FORTTEMP must know exactly how much temporary space is required by each assembly. The Pass 1, Pass 2, and Pass 3 local temp areas are defined on top of each other to save space.

X1X2DCBS

Contains definitions of M:X1 and M:X2, the scratch file DCBs. This assembly is not included in the RBM version, because the loader satisfies references to M:X1 and M:X2 with DCBs that have the proper defaults. This is explained in Chapter 4, under Open Files (4.4.1).

P2P3ROOT

Contains a table of constants referenced by Pass 2 and used in Pass 3. As its name implies, P2P3ROOT properly belongs in an overlay path that is common to Passes 2 and 3, but not Pass 1. (Overlay Structure Details, 4.5.18, explains this.) For the most part, Pass 2 sends Pass 3 the address of some constant in P2P3ROOT, and then Pass 3 uses the value of the constant. These constants are the library links (i.e. the names of library routines that Pass 3 must externally reference, such as 9SQRT, 9RTOI, F:108) and also text strings for comments that are to appear on the object listing (such as "COMPARE WITH ZERO"). P2P3ROOT also contains one static list, the UNITLIST, that helps the compiler decide when to reference DCBs for certain FORTRAN I/O units, such as 105 and 108.

PASS1F PASS1)

This is the overall control for Pass 1. It initializes before and cleans up after the source program has been read. It does the same for each statement. It scans the label field of each line, and then calls STASCAN to scan the statement. It prints the source listing and most of the diagnostics. And it contains the TRY-Fail procedures (see 2.7). Under its control, Pass 1 reads the source program and outputs a digested version thereof that is suitable for code generation by Pass 2. It also builds the Symbol Table and other lists giving the attributes of names and other constructs used in the source program. (This includes labels, constants, DO loops, arithmetic IF statements, etc.) PASS1F is coded in POPs.

STASCAN

Scans a part of the source program looking for a statement. If there was an S in column 1, it calls SINCOLM1 to handle this. STASCAN either finds a legal statement and files code for it on the CODELIST (usually), or it diagnoses the construct as illegal. In STASCAN is the knowledge of the FORTRAN language at the statement level. For example, it knows that a REWIND statement consists of

REWIND expression

without knowing precisely what an expression is. (That is left to EXSCAN.) STASCAN is coded in POPs.

EXSCAN

Scans for an expression. It has several entry points such as integer expression, logical expression, any type expression, etc. EXSCAN is coded in POPs and is recursive (i.e. it can call itself). This is because the definition of expression is recursive; one possible element of an expression is another expression enclosed in parentheses.

CONSCAN

Contains the POP portions of the routine for scanning constants. (The machine language portions are in PASS1ML.) CONSCAN recognizes integer, real, double precision, complex, double complex, logical, Hollerith, and hexadeciaml constants. It scans their syntax, converts them to an appropriate internal representation (e.g. Sigma floating point for real), registers their values on an appropriate constant list, and returns a pointer to the constant.

PASS1ML

Contains all of Pass 1's machine language code. This includes part of the TRY/Fail mechanism, part of constant scan, the input character processing routines, several POPs peculiar to Pass 1 (e.g. ERC, SOP, FEX), and miscellaneous other subroutines.

SINCOLM1

Scans lines with S in column 1, that is, in-line assembly lines. These lines are permitted only if the S option has been specified on the control card. (See Overlay Structure Details, 4.5.18, for the reasons behind this.) SINCOLM1 is coded in POPs. It also contains a large table of Sigma machine op-codes and their characteristics (i.e. what kind of register, index, and address fields may or must be used with them; this is diagnosed very carefully). SINCOLM1 contains its own error messages, whereas the messages for all of the other assemblies in Pass 1 are in STASCAN.

PASS2F PASS2)

Generates object code. Reads the output from Pass 1 (which is a digested representation of the source program) one statement at a time and produces code. Input is read from the Input List (which may have been spilled to a scratch file during Pass 1, when it was called the Output List). Output goes onto the Code List, from which it eventually gets to the Output List. As the code for each statement is moved from the CODELIST to the OUTLIST, it passes through a routine called Assemble Code, which operates like the first pass of an assembler; it keeps a location counter and defines labels. Since this assembly pass is not done until the end of the statement, code can be generated out of order and shuffled around, which it frequently is. It should be noted, however, that Pass 2 always shuffles the code around such that it appears in the object listing in the same order that it will appear in memory.

ALLOCATE

This is an "interlude" (as opposed to a pass) that is performed just after Pass 1. ALLOCATE decides what location each variable is to be allocated to. It does this by examining the Symbol Table along with the inforamtion from

COMMON and EQUIVALENCE statements that has been accumulated by Pass 1 on the BLCOMLST, LBCOMLST, BLNAMLST, and EQUIVLST. It can produce a wide assortment of diagnostics about misuse of COMMON and EQUIVALENCE. At the end, all scalars and arrays (including dummies) have been given locations, and these locations are in the Symbol Table.

PASS3F PASS3

Contains the overall control of Pass 3. This includes printing the introductory part of the object listing, allocating the temps, and printing the final part of the object listing. It calls GENLIT to generate the literals, OUTPROG to output the bulk of the object listing and the binary object program, and SUMMARY to print the final summary. Also contains miscellaneous routines used in Pass 3, such as printing routines, the Pass 3 error exits, and Evaluate Pointer, which analyzes pointers generated by Pass 2. Finally some quote strings and parametric values are defined in this assembly.

GENLIT

Generate Literals. This passes through the various constants lists, determining which constant have actually been used in the generated code, and registering these on the Literal List (LITERLST). It does this with an overlapped search, in order to double up on literals whenever possible. It sets a flag to indicate whether the literals must begin on a double-word boundary.

OUTPROG

Performs the second pass of assembly on the object code produced by Pass 2. Produces the object listing and/or binary output, as requested. The object listing is produced in OUTPROG itself; binary output is produced in OUTBO, driven by OUTPROG. Input is read from the INPUTLST (which may bave been spilled to a scratch file).

OUTBO

Creates the binary output for Pass 3. Binary output can go to either BO or GO or both. Each time OUTPROG has a word it wants output, it calls OUTBO. OUTBO understands the Sigma binary language and breaks up the output into appropriate records. OUTBO is coded partly in POPs and partly in machine language.

PRINTCON

Called by OUTPROG in Pass 3 to list constants on the object code listing. It goes to some trouble to list them in a "nice" readable fashion, which may depend not only on their type built also on their value. The constants it prints are usually literals, preceded by an equal sign (the equal sign is actually printed by OUTPROG), but sometimes PRINTCON is used to print immediate values. PRINTCON receives a pointer to one of the constant lists. Sometimes the mode field of the pointer is also important in deciding how to list the constant. For example, zero and .FALSE. are the same constant on the

same list, but the mode determines how it should be listed. PRINTCON is coded partly in POPs and partly in machine language.

SUMMARY

Prints the compilation summary (usually) or part of it (if the PS option was specified) or none of it (if NS was specified). The summary includes the alphabetic symbol table, the label table, the various functional groups (e.g. EXTERNAL SUBPROGRAMS REQUIRED), and the final statistics.

PATCH #PATCH

This and the remaining five assemblies are all small, special assemblies included only to facilitate patching of the compiler under various operating systems. This is explained in Overlay Structure Details, 4.5.18. PATCH is a block of protected storage (i.e. CSECT 1) for instructions, as opposed to temps, that are part of patches.

PATCHT #PATCHT

(See PATCH.) This is a block of unprotected storage (CSECT 0) for temps, as opposed to instructions, that are used by patches.

PATCH1 #PATCH1

(See PATCH.) This is a no-size assembly that provides DEFs of some symbols in the Pass 1 overlay segment, so that those symbols may be referenced by patches. The symbols are equated to PATCH and PATCHT in the root segment.

PATCH2 #PATCH2

Same as PATCH1, but for the Pass 2 overlay segment.

PATCH3 #PATCH3

Same as PATCH1 and PATCH2, but for the Pass 3 overlay segment.

REF no external DEF

This is also a no-size assembly that provides REFs for all the assemblies in the Pass 1, Pass 2, and Pass 3 overlay segments. Without these REFs, patches in the root segment would not be able to refer to those other assemblies. See Overlay Structure Details, 4.5.18.

FORTCOMP

The contents of this module are included with those of many other compiler modules through their use of the Meta-Symbol directive SYSTEM FORTCOMP. FORTCOMP is a collection of definitions of parameters and Procs used throughout the compiler. These include the following:

- Registers: the names used for the sixteen Sigma hardware registers, such as X1 EQU 1 (see "Register conventions in the Interpreter", 1.9.2.3).

- Type (Mode) codes: such as INTG EQU 1

- Character set: the names used to identify members of the FORTRAN character set, expressed as hexadecimal integers, as in LETTERA EQU X'C1' and period EQU X'4B'

- External References: REFs of often used symbols, like the recursive exits (EXIT, EXITML, EXITTRUE, etc. -- see 1.9.3.3); the CENTRALs (CENTRALO, IOCENT etc.); EATEMP; and a few subroutines (WMODXIML, etc.).

- Special Address Parameters: special list-addressing parameters like WO, P2, and P3IND (see 1.9.1.2); "PAGE", used with the PRL Pop, and "LEFTMARG" used with PBL.

- POP Procs: Procs that assemble POPs. These are described in detail in 1.6.3. Each POP name is also REFed for use as the value of the POP's "op code".

- Temp Procs: the Procs TEMP and DTEMP used throughout the compiler to allocate storage (in FORTTEMP) for the compiler's local context.

1.4 Lists

1.4.1 Dynamic Lists

Most of the compiler's processing and information saving is done on lists. Lists are used both to accumulate data (e.g. the Symbol Table) and to manipulate it (e.g. the Work List). For the latter purpose, lists are commonly used instead of registers, because this facilitates the recursive use of subroutines. When information is processed in registers (as in the run-time object code, for example), much caution is required to avoid conflicts between subroutines. Either they must use different registers, or the calling routine must save the registers it does not want destroyed, or the called routine must save the registers it destroys. And this is impossible if a subroutine calls itself. None of this is a problem when using dynamic lists. As long as each level of subroutine takes off whatever data it puts on, each can use the lists without interferring with information previously put on, and without being interfered with by information put on at lower levels. This is true even when the different levels are actually the same subroutine. The other way in which the lists facilitate recursion is by stacking up the return addresses for subroutine calls. This is done on the

Exit List. All recursive subroutines exit to the place specified by the last word of the Exit List.

1.4.2 How lists are stored

The compiler does not use threaded (linked) lists. Each list occupies a contiguous area of storage. The reasons for this are:

1. Threaded lists would take up more space, especially since we often access data from either end of the lists, so they would have to be threaded in both directions.

2. Items can easily be addressed by relative position within the list. The Nth item on a list with entry size 3 is 3N-2 words from the base.

3. Large blocks of data can be directly accessed. For example, we read the source images directly into 20-word chunks of the Input List.

If the lists are not to be threaded, then they either must have fixed locations and sizes or they must be dynamically allocated (that is, slide around in memory to avoid bumping into one another). The reasons for using dynamic lists rather than fixed ones are:

1. Fixed size lists for unknown inputs are odious. (The Packard Bell PB-440 FORTRAN Reference Manual tactfully refers to them as "cruder and more rigid storage techniques".) Each one of them places some sort of restriction on the source program which some user is eventually going to encounter, to his displeasure.

2. Use of available memory is less efficient. One list can overflow despite the fact that there is lots of unused space being held for other lists.

3. On certain lists, we intentionally put things on at one end and take them off at the other. The effect of this is that the list "moves" through memory. This is difficult with a fixed location list.

4. One of our dynamic lists is the input buffer. This enables us to allow any number of continuation cards, without penalizing programs that do not use them. The inter-pass scratch files are also dynamic lists, which are kept in memory (to avoid the use of actual scratch file devices) as long as there is room for them.

Associated with each list are four parameters that determine its location, size, and status. These are the Base, Start, Top, and Bottom. (There are also two others, Code and Flags, which are described later.)

The Base is the absolute beginning of the list. It points to the word immediately preceding the word that is physically first in the list. All pointers to items in a list are relative to the Base.

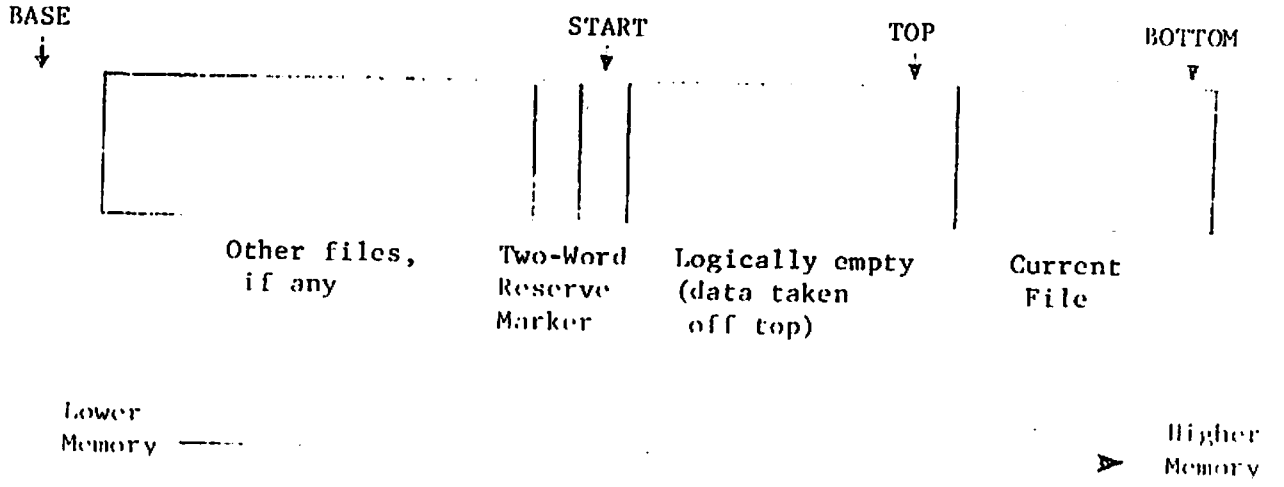
The Start is the beginning of the current file. If more than one file exists on the list (i.e. the list has been reserved), the Start points to the word immediately preceding the word that is physically first in the bottom (i.e. last or current) file. Most operations deal only with a single file of data and thus treat the Start as the important boundary. Note that if the list has not been reserved, the Start is the same as the Base.

The Top is significant only if data has been taken off the top (or beginning) of a file. In this case, there is an empty space between the previous file and the "true" beginning of the current file. The Top, therefore, points to the word immediately preceding the word that is logically first in the bottom file. It is not common to take data off the top of a list, so the Top is usually the same as the Start.

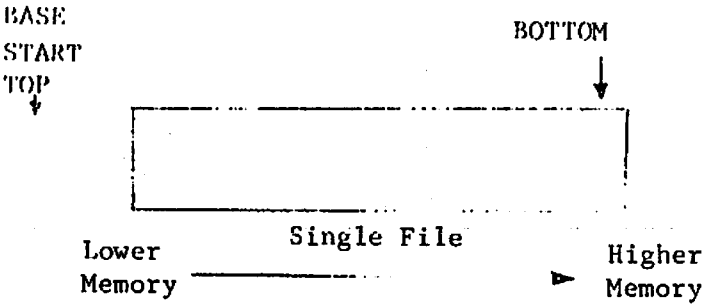
The Bottom is the last word on the list (not the preceding word). In the typical case (no files, nothing taken off the top), the Bottom is the only one that actually lies within the list; the others are all equal and point to the wrod just prior to the beginning of the list.

The "file mark" for lists is called a reserve marker. It is created by the RSV (Reserve) POP and consists of a two- word entry containing the Start and Top (relative to the Base) of the previous file. When a RLS (Release) POP is performed, this two-word entry tells how to restore the official Start and Top values for the newly-uncovered file. (There are also other POPs that reserve and release lists as part of their operation.)

A slight confusion results from the fact that the lists are arranged forward in memory, so the Bottom of the list lies towards the top (high) end of memory, and vice versa (assuming that you think of loaction 00000 as the bottom of memory). As a compromise, the following diagram is drawn sideways, so that you can think of memory either way. The diagram shows the locations of the Base, Start, Top, and Bottom of a list.



On the other hand, a typical list, which has not been reserved and has not had data taken off the top, looks like this:



The list parameters (Base, Start, Top, and Bottom) are kept in fixed tables in the assembly LISTDEFS. The Bases are all together in one table, followed by all the Starts, followed by the Tops, followed by the Bottoms (followed by the Codes, followed by the Flags). The beginning of the Base table is called BASE, and similarly for the others. This means that the Base of the Nth list is at location BASE+N; the Bottom is at BOTTOM+N, etc. Since lists are commonly dealt with by their list number (position in the order of lists), this value can be loaded into an index register and the various parameters easily addressed.

Each list also has a parameter called the Limit. This identifies the last word than can be used by a list before it overflows into the next list. There is no special table for Limits; the Limit for each list is simply the Base of the next list. This means that the label LIMIT can be defined at BASE+1, so the Nth list's Limit is likewise at LIMIT+N.

Some of the list parameter words have labels defined on them; others do not. The use and dynamic allocation of lists proceeds automatically, without having to address specific list parameters by name. Occasionally, however, the coding in the compiler needs to directly address these parameters. For example, the bottom (most recent) item on a list can be accessed by addressing indirectly through the list's Bottom. For example,

```
MPO    *WORKBOT
```

(Getting another level of indirect addressing, that is, indirect through the value on the bottom of the list, is only possible on the Work List and Param List and is handled in a special way. This is described in 1.6.2.)

1.4.3 Static Lists

There are several static lists, so called because they have fixed locations and sizes and do not participate in the usual reallocation of memory. These lists usually contain tables of information that are built into the compiler; e.g., the Intrinsic List (INTRILST) contains the names and characteristics of the intrinsic functions. They are made into lists so that they can take advantage of some of the automatic list procedures, such as being searched and having pointers to them. Since they are lists, they have the same parameters in LISTDEFS as the dynamic lists (Base, Start, etc.). The difference is that these parameters have fixed values assigned to them at assembly time, so they are neither initialized nor reassigned during completion.

The static lists are not located in general list memory either. In fact, if they contain fixed information (such as the Intrinsic List), they do not even have to be in unprotected storage. The Intrinsic List is contained in the assembly EXSCAN. Therefore it disappears when Pass 1 is finished. No check is made for this; it is incumbent upon the compiler not to use a static list that is not available. Other static lists, such as the Index Register List, have a fixed size and location, but not fixed data, and so are contained in unprotected assemblies (e.g. FORTTEMP). The list parameters, however are always in LISTDEFS.

There are also a few fake lists. These have paremeters in LISTDEFS, but only the Code and Flags words are used. They exist so that the compiler can work with pointers to things that are not on lists and still use the normal procedures, some of which assume that things are on lists. For example, an element in an expression might be a constant (on a constant list) or an operation like multiply (on the Plex List) or it might be a temp, to which a value has previously been assigned. A temp is not on a list; it merely has a number. It is the Nth temp. Using the fake list TEMPNTR, with an address of N, the compiler can think that this is the Nth item on the TEMPNTR list until it gets to a place that really needs to know what to do with this.

The ZEROLISTis a special fake list. It allows a simple address (i.e a PZE, with nothing in the upper bits) to look like a list pointer also.

1.4.4 List Pointers

Items on lists are normally identified by pointers. A list pointer contains, in the address field, the item's position in the list (relative to the Base). It usually contains the list number in the first two hex digits and the mode (if applicable) in the third hex digit. The position in the list is relative to the Base and remains the same even though the list may move around in memory. However, if the relative positions within a list are changed, a pointer is no longer valid. This can happen if a list is sorted (SORT POP) or flipped (FLIP POP). It can also happen if words are taken off the top of a list that has not been reserved, because in this case, the empty space created is at the beginning of the list, so the Base and Start are automatically moved up to follow the Top. The list is, in effect, shortened from the beginning end, and every item is now a different distance from the Base, so previous pointers are invalid. That this does not cause trouble is indicative of how few lists have information taken off the top.

Pointers are described in greater detail in the section on Pointers and Pseudo-Pointers, Section 1.11.

1.4.5 Searching and Registering; Entry, key, and offset size

It is often necessary to recognize subsequent appearances of the same construct in a FORTRAN program. For example, when a symbol first appears, it is added to the Symbol Table (the only list whose name does not contain the word "list"). Later appearances want to refer to the same entry on the Symbol Table. To "register" something on a list means to makes sure that it is on the list and create a pointer to it. This involves searching the list first. If the item is found, a pointer to it is returned. If the item is not found, it is added to the list, and a pointer to the new entry is returned. This is a very common procedure in the compiler.

Fortunately, most list entries are of fixed size. (Those of variable size must be handled specially.) For example, a Symbol Table entry is six words long. Only the first two words of a Symbol Table item are significant for searching, however, because they contain the eight-character name of the symbol and that is what we are searching for. Therefore, each list must specify the length of an entry (entry size), the number of words that are significant for searching (key size), and the relative position within an entry of the key words (offset). These values are all contained in the Code parameter value in LISTDEFS (which was mentioned earlier). As it happens all the lists currently in the compiler have zero offsets; that is, the key words are the first ones in the entry. However, the DOLIST is sometimes searched with offset of 2 (using the SERK POP).

The Code word also contains the list number in the first two hex digits (sometimes used for pointer identification) and the mode in the third hex digit (necessary for building pointers with the correct mode field).

A key size of zero is used for label lists. These lists have the label value in the lower 20 bits, and this is what should be searched for. The upper bits contain flags which must be ignored when searching. Therefore, a key size of zero means a key size of one but with just the label value field (lower twenty bits) searched.

Sometimes it is desirable to be able to search a list with different key or item sizes. For example, in GENLIT (General Literals, in Pass 3), the Literal List is searched with a variety of sizes and keys. For double complex constants, it is even searched with a key size that is bigger than the item size. (A double complex constant might overlap the imaginary part of one constant and the real part of the next constant.) For this purpose, there is a SERK POP (Search with Key). This allows you to specify the "Code" word to be used during the search, rather than using the Code word associated with the list.

1.4.6 A few basic lists

A few of the lists are so basic to the compiler that they deserve some special attention. Whereas most operations with lists require the list to be identified, these lists are automatically associated with certain basic compiler operations and often are not specifically named. As such, they are almost analagous to registers (for the compiler) while the ordinary lists are more analogous to memory locations.

The Work List is the principal list in the compiler. Some 65 POPs automatically know that they are to use this list. The POP language can be thought of as having a very limited number of registers, and the Work List is the main one, the accumulator. (There is also an A register, and the Param List and Exit List operate somewhat like registers.) Loads, stores, tests, moves, arithmetic, and other manipulation all take place on the Work List. The section on POPs discusses the various operations that can be performed on the Work List.

Since it is essentially a push-down stack, it is often desirable to get at items on the Work List other than the most recent (i.e. the one on the bottom). For this reason, special symbols are defined for the last five items on the Work List. The bottom one is called W0, followed by W1, W2, W3, and W4. Thus, for example, FET W1 fetches the second item on the Work List and appends it to the bottom of the Work List. It therefore becomes W0, what was W0 becomes W1, and what was W1 becomes W2, and W0 is now the same as W2. Some rather special finagling is necessary to allow the POPs to address items other than the bottom. (The bottom can be addressed indirectly through BOTTOM+WORKLIST, where WORKLIST is the list number for the Work List). This is also explained in the section on POPs.

The PARAMLST (or Parameter List) is used for passing arguments (or parameters) to subroutines. Of course, arguments are also often passed on the Work List, or in other places, but the Param List is set aside for parameters that are going to be in force for quite some time and do not want to be distrubed by manipulations that will have to take place on the Work List. For example, within Expression Scan, the permissible types are carried along on the Param List, so that they can be checked each time an element is found. The ADR (Address to Param List) POP is usually used to place values on the Param List. As with the Work List, the first few items on the Param List are directly addressable by using the special symbols P0, P1, P2, and P3. In addition, since the Param List is often used to hold the addresses of arguments, and therefore wants to be addressed indirectly, the first few items on the Param List may also be indirectly addressed by using the special symbols P0IND, P1IND, P2IND, and P3IND.

The Exit List is seldom specifically referenced. It automatically takes care of stacking up return addresses from subroutines, so that the compiler may function recursively without any special checks. The JRS (Jump Recursively to Subroutine) POP always stacks up its return address on the Exit List, and the various exit points (e.g. EXIT, EXITTRUE, etc.) always pull one return address off and return there. There is a peculiar section of code in the DATA

statement (Expand DATA Pair Sequence) that requires two exit lists, so it uses the Param List as an alternate exit list, and copies them back and forth in order to have the right stack of exits on the Exit List at the right time.

The Exit List is also used to maintain the answer flag (which is either TRUE or FALSE and is set and tested by a variety of POPs). The address field of the current (bottom) word on the Exit List contains the return address for getting back to the upper level. The sign bit contains the answer flag for use as this level. Upper level's answer flags are preserved in the sign bits of Exit List items previously stacked up on the list.

The Plex List may also be considered one of the "built in" lists, since there are several POPs that automatically use it, either for building or unraveling plexes. This is described in the section on Plexes, 1.10.

1.4.7 Scratch Files

Included among the dynamic lists are the inter-pass scratch files. These are kept in memory as long as possible, to avoid using an actual scratch tape. If memory is used up, the output data can be dumped to a scratch file. The output from Passes 1 and 2 is kept on the OUTLIST (Output List). It is not used by Pass 3. The Output list is an ever growing list that starts at the low end of working storage and pushes everything ahead of it. (Actually, everything is arranged in such a way as to hopefully minimize the amount of pushing necessary, as described in the following two sections.)

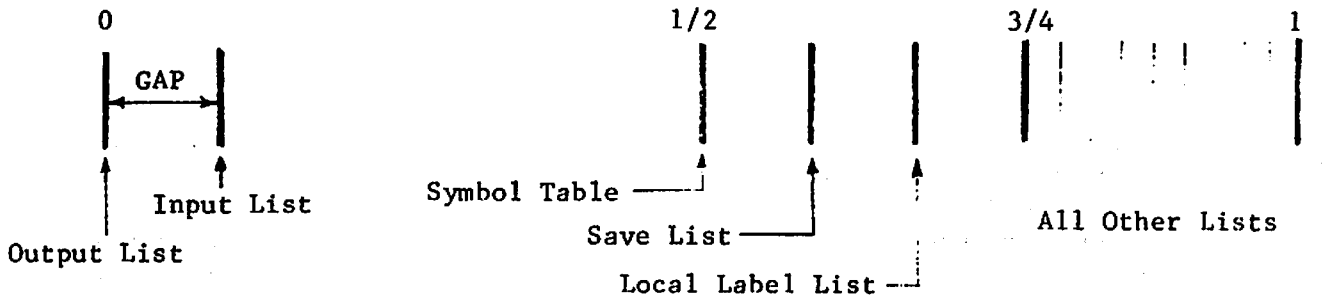
After Pass 1, the Output List becomes the Input List for use by Pass 2, and the Output List starts afresh. The same thing happens after Pass 2. In Pass 1, the Input List is used for the source input string. It is incremented in chunks as a new source card is read and tacked onto the bottom of the list. As the source lines are processed, they are taken off the top of the list, and the Input List thus moves through memory, hopefully staying ahead of the onrushing Output List. It seems likely that when no LO is requested, the Input List will keep ahead of the Output List fairly easily. Of course, when an object listing has to be printed, the source images have to be passed along on the Output List, so it constantly overruns the Input List, but when an object listing is being printed, time is not of the essence anyway. Speed should be optimized for the cases that really need to run fast.

Typically, the Input List contains only two cards' worth of data. It needs the second one to check for continuation. It will expand to more than this only if there is a continuation card and it has to check for another one. In this way, it can allow any number of continuation cards, without using any space for them when they are not present (which is most of the time). Note that the forward order, non-threaded character of the lists makes it possible to read cards directly into the Input List and to print the information contained therein by using the PAC POP (Print Alphanumeric with Count).

1.4.8 Initialize Lists (INILISTS)

This routine is called during initialization of the compiler (in FORTRAN4). It sets up the parameters (Base, Start, Top, Bottom) of all the dynamic lists. It also sets up the inital value of the Input Offset, dependent on the Base of the Input List. (See Next Input Character in 2.2.3.) And it puts one "dummy" word on the Exit List, so that there is a place to put the answer flag for the zero level of recursion.

Initially, the four parameters of each list are equal (i.e. the list is empty). They are distributed throughout the available working storage as shown in the following diagram.



0 is the beginning of list memory. Its address is contained in PBOTLMEM, previously set by Get List Memory (GETLMEM, in MERRIMAC). 1 is the top of list memory, and its address is contained in PTOPLMEM. (Note the difference in terminology that the top of a list is towards the bottom of memory.) 1/2 is the midpoint of list memory, and 3/4 is three-quarters of the way from the low end to the high end.

The Input List is expected to move through memory as input is appended to its front (bottom) and removed from its back (top). The Output List is expected to grow continuously and fairly rapidly. Therefore, these two lists are initially given half of memory. The Input List rolls along like an inchworm, and the Output List builds up behind it. Theoretically, the Input List moves faster than the Output List. If this is not true, they will constantly bump into each other. When the Input List hits the Symbol Table, the Input List is moved down to its starting position, just above the Output List. This is shown on the diagram as GAP, which is currently 100 words.

The Symbol Table, Save List, and Local Label List are known to use quite a lot of space. The theory here is to give them some extra breathing room, by giving them each 1/12 of memory. They are evenly distributed between the 1/2 and 3/4 positions.) All of the other lists are evenly distributed in the remaining quarter of memory at the high end.

This was an early guess as to an efficient allocation of the lists. Very little study has ever been done to measure its suitability. It seems likely that it could be improved noticeably in some of the following ways.

1. Of prime importance, there are too many lists. Since the lists do work recursively, a lot of operations that take place at different levels could

be done on the same list, and they would not interfere with one another. In addition, there are certain lists that are used only in certain passes, and these could be doubled up with one another. Note that this must be done with extreme care, not only to avoid conflicts, but also because lists may have different entry or key size, offset, mode, or flags. Still a large number of lists have no interesting characteristics at all. They are just scratch lists.

2. There are probably other lists that could use extra breathing room. Perhaps the Equivalence List and the COMMON lists, or the Array List, or the Group List, almost certainly the Plex List. If this were shown to be the case, these lists should be down in the 1/2 to 3/4 range.

3. Then, in general, the lists should be arranged in decreasing order of their propensity to grow large. This is because the lists at the high end have to push all the lists below them to make room, and this should happen as seldom as possible. The lists that have to push should be at the low end where they will not have as much to push.

The above is a description of the algorithm used by INILISTS when it has plenty of list memory to work with. In case there is only a small amount of memory available initially, then a much simpler scheme is used: the lists are all "piled up" at the low end of the available memory; i.e., the Bases, Starts, Tops, and Bottoms of all lists are set equal to the contents of PBOTLMEM.

Whichever algorithm is used, INILISTS completes its work by creating one word on the Exit List so that there will be an Answer Flag (see "True/False Feature:, 1.6.1) for recursion level zero.

1.4.9 Reassign Memory REASSIGN

The previous section showed the initial distribution of the lists. When these lists bump into one another, a variety of techniques are used to make room for them. First we must define some terms. An ordinary list is anything other than the Input List of Output List (which are handled differently). Moving "down" means towards lower memory, or towards the top of a list.

When an ordinary list needs space, we attempt to move it and all the ordinary lists below it down into the space above the Input List. This is done without removing the spaces between these lists. Most lists quickly establish an appropriate amount of breathing room, and it would be costly to keep taking this away. This is only done if necessary. If there is not enough room above the Input List, there may be below the Input List. If so, we can slide the Input List down and then proceed with moving the other lists down.

If the combined space above and below the Input List is insufficient, then we are forced to remove the gaps between some of the ordinary lists. We do this by squeezing them all together towards the high end of memory. This is called pressing. Since the lower lists are supposed to be the most volatile, we try first to press only the lists above the list needing room. If this does not provide enough space, then we also press the ordinary lists below this list. This results in all the ordinary lists being pressed up against one another at the top of memory, with no gaps. Since the Input List has already been moved down, we now have all the unused memory in one chunk between the Input List and the first ordinary list. Again we try to move down into that space to obtain room. If this is not enough, we may be in trouble. There are two possible courses of action left. The Output List is an inter-pass scratch file, which is kept in memory as long as possible. It is no longer possible. Therefore, at this point, the Output List is dumped to an external file. This is done in 200-word records, until there are fewer than 200 words remaining in the Output List. Then the OUTAPETG (Output to Tape Trigger) is set. This means that from now on, the Output List will not be allowed to grow in memory. Periodically it will be checked and, if greater than 200 words, a record will be written out.

This will usually gain the required space, unless the Otput List has already been dumped out, in which case it will probably gain no space. We try moving down the Input List and the lists below the list needing room, and if this still does not get us enough space, we call GETAMEM (in MERRIMAC) to try to get additional working storage from the Monitor. Only under UTS is it possible to get more. Under RBM and BPM, we have already taken everything there was upon initialization. (See Get List Memory, 4.3.1, and Get Additional List Memory, 4.3.2.) It would seem more efficient under UTS not to wait quite this long before requesting additional memory. The compiler gets into noticeable time-binds as it approaches the situation where all the lists are tight up against one another. This is not too bad if it happens only once, when memory is truly running out. But it is probably wasteful for it to do so every time it gets one additional page of memory. One can envision this increasing compile time significantly.

After asking for additional working storage (which comes at the top), we repeat the same process, starting from pressing the lists above the one needing room. If all this fails to get the required space, then we branch to OVERFLOW to print an error message and abort the compilation.

If the list that needed room was the Input List (instead of an ordinary list), then the situation is similar except that there are no ordinary lists below it. Therefore, we first try moving the Input List down. Then we press all the ordinary lists up. Then we dump the Output List to a scratch file and try moving the Input List down again. Then we ask for more working storage and, if we get it, repeat the process starting from pressing all the ordinary lists up. Otherwise we overflow.

If the list needing room was the Output List, the procedure is a little different. First we try to move the Input List up, into the space between it and the first ordinary list. Furthermore, if there is enough room, we do not just move the Input List up exactly the amount required; we move it up about 20 words additional, on the assumption that the Output List may have more words added to it before anything gets taken off the Input List. This may avoid having to move the Input List several times. If there is not room enough above the Input List, then all the ordinary lists are pressed up and we try again to move the Input List up. Then we dump the Output List to a file. If that has already been done, then we ask for additional memory and repeat from pressing the ordinary lists up. If there is no additional memory to be had, we must overflow.

Note that any time the Input List is moved (either up or down) the Input Offset (INPOFSET) has to be adjusted to reflect the virtual position of the zeroth character in the source input string. When the Input List moves down (which is typical), the Input Offset is decreased (every character, including the zeroth, is now in a lower address). When the Input List is moved up, the Input Offset is increased. See Character Scanning, 2.2, for a description of the counts associated with the source input string.

There are two further facts of interest pertaining to the reassignment of lists. These are two occasions when list parameters are changed unexpectedly. These do not take place in Reassign Memory, but in the TOT POP (Take Off Top).

1. When information is taken off the top of an unreserved list, the Base and Start are automatically moved up to coincide with the Top. This means that there is no logically empty space at the low end of the list; these words are freed up for use elsewhere. It also means, however, that pointers to the list are no longer valid (since the Base has moved, but not the information in the list). Therefore, information should never be taken off the top of lists that have pointers (most lists do not).

2. When information is taken off the top of a list that has been reserved, there is a gap created between the current file and the previous files. However, when the last (bottom) word is taken off the top, so that the current file is empty, the Top and Bottom are both adjusted down to coincide with the Start, thus the freeing space that is now all logically empty.

1.4.10 Freight Trains

The normal mode of operation of a dynamic list is as a push-down stack. Items are put on the bottom and then taken off the bottom, so the list expands and contracts, but remains more or less in one place. The Base will not move unless the list bumps into another list and has to be reassigned, and after this happens a few times, the list will have acquired the breathing room it normally needs, and it can settle down. Some of the lists grow continously, such as the Symbol Table and constant lists. These can be a problem and some special provision is made for them in Initialize Lists.

The worst problem is caused by putting information on the bottom of a list and taking it off the top. In this case, the Base automatically slides up to follow the Top, and never retracts to where it was. Data keeps being put on the front and taken off the back, and the list moves through memory, like a freight train. This has disastrous effects on the peaceful allocation of the other lists, because every time this list bumps one in front of it, this list and all the ones behind it are moved down, maintaining the large unnecessary gap below this list. Eventually, the lower lists will get down far enough to require pressing together of the upper lists, all because of an unneeded and ever growing gap beneath the "freight train".

The Input List is expected to be a freight train, and it is specially located beneath the ordinary lists, so that it can constantly be moved down without disturbing anything else. The only other lists that have a tendency to become freight trains are the Alpha List (see Saving Alphanumeric Characters, 2.5) and the CODELIST. Characters are piled onto the bottom of the Alpha List and pulled off the top (i.e. it acts like a queue rather than a push-down stack). Therefore, in the FORMAT Statemment, we reserve the Alpha List to keep it from moving. With the list reserved, the Base will not follow the Top. Rather, when the list is emptied, the Top will revert to the Base.

Words are moved onto the bottom of the CODELIST for interface between the passes. Then they are removed and placed on the OUTLIST. In Pass 2, these are removed one at a time from the Top of the list by Assemble Code. If there were not a reserve mark on the code list, it would be a freight train.

1.4.11 The List Procs

The list parameters are all defined in the assembly LISTDEFS. They are defined using a group of Meta- Symbol Procs:

LIST (BASE) (START) (TOP) (BOTTOM) (LIMIT) CODE (FLAGS)

The Procs shown in parentheses are optional. Every list must have the LIST and CODE procs specified.

As explained previously, the list parameters are stored with all the Bases together, all the Starts together, etc. Therefore, the first thing done in LISTDEFS is to reserve the areas for these parameters. These areas are labeled BASE, START, TOP, BOTTOM, and FLAGS. The CODE area begins right after the FLAGS area and is generated as we go along, by the CODE proc. There is no LIMIT area. LIMIT is simply equal to BASE+1. Note that there is no problem in Meta-Symbol about having procs and labels with the same name.

The other initialization necessary is to set NL (Number of Lists) equal to -1, so that when it is first incremented, it will be zero.

Then, for each list, there must be a LIST proc reference, with the list name in the label field. This defines the list name to have the corresponding list number. Henceforth, each list will be referenced primarily by its number. If any part of the compiler wants to reference the Base, Start, Top, or Bottom of a particular list by name (e.g. WORKBOT or INPUTBAS), then the corresponding proc can be used to give that parameter a label. Otherwise these procs are not needed. The area has already been set aside for them and the list has a number. That is all that is needed to access any of the parameters.

The CODE proc generates the Code word and increments the location counter by one. The LIST proc sets LN to S-CODE and gives a diagnostic if this is not one greater than the previous LN. This verifies that the lists are progressing in consecutive order and that each list has both a LIST proc and a CODE proc. If the CODE proc is used with no arguments in the operand field, it generates a standard Code with key size and item size of one, offset of zero, and UNKNOWN (zero) mode. These values may be specifically set up, however. Again, if the CODE line has a label, then it will be possible to reference that list's Code word directly (e.g. SYMTBCOD).

The FLAGS proc is only needed on lists that have traits to be tested. It generates (in the FLAGS area) a word with the list number in the upper eight bits, the mode in the next four bits, and the traits specified in the lower twenty bits.

The static lists illustrate another feature of the list procs. If the BASE, START, TOP, or BOTTOM references have a label in the operand field, then that defines where the list is located. For example, the Bottom of the Intrinsic List is at INTRIHI (which is in EXSCAN), and the Top, Start, and Base are all at INTRILOW. The procs will put this value into the appropriate parameter, and it will never be changed by Reassign Memory. Note, however, that just

putting a value there is not enough to keep the list from being reassigned. The value LASTLIST must be equated to the last dynamic list, before setting up the static lists. This is the value used by Reassign Memory to determine the end of the dynamic lists.

The fake lists have much the same structure. Their only important parameter is their CODE word, but of necessity, the area for their other parameters is nonetheless reserved.

There is a fair consistency used in labeling the list parameters. The list number always ends with the characters LIST or LST (except for the Symbol Table, which is SYMTABLE). The Base ends in BASE or BAS. The Start ends in START, STRT, or ST. (There are very few Starts defined; it is curious that they have more variation than any other name). The Top always ends in TOP. The Bottom always ends in BOT. The Limit ends in LIMIT or LIM (there are also very few Limits defined). The Flags word never has a label. It could, but there is little reason to access directly the Flags word of a specific list. If you know what list it is (such that you can address it by name), you presumably know what its traits are. The Flags word is usually used when you have a list number and you want to know what its traits are. The reason that some of the Code words have labels, is that the Code word can be compared with a pointer to find out if it is a particular list pointer. The Code word (if it has a label) ends in CODE or COD.

1.5 Table of Lists

Following is a table of the lists in the compiler. For each list, the following information is given (or however much is applicable):

1. The symbolic name and the descriptive name of the list. Static lists are indicated as such.

2. If applicable, the entry size, key size, and mode of the list. As described above, these are defined in the Code parameter for the list. For many lists, the entry and key size are not important, because nothing is done that requires them (e.g. SER, SORT, JPN). When these are not specified, they are given default values of 1. No list has an offset other than zero, so they are all defaulted to zero. On most lists the mode is also unimportant, and so is defaulted to zero (i.e. Unknown). Note, however, that for the String Constant List the mode is important and must be Unknown, so it is specified as such. A key size of zero is used for lists that need to be searched or sorted based on only the lower twenty bits of the key word. Thus a key of zero really means one, but only the low twenty bits. This is used primarily for labels.

   Many lists have a variable size entry; these are usually defaulted to entry and key of one. Other lists are used in some places with fixed length entries and other places with variable length (e.g. the SAVELIST). If there were different size fixed length entries in different places, one of them would be chosen as the "official" size in the Code word, and the others would have to be handled specially (e.g. searching with SERK instead of SER). Obviously all variable length entries must be handled specially.

3. If applicable, the traits defined (in the Code word) for the list. Most lists do not have traits. Symbol Table pointers have traits, but they are defined in the ID word (YSYMBID) of the SYMTABLE entry, rather than in the SYMTABLE Codel. PLEXLIST traits are similary built into the plexes themselves, rather than into the PLEXLIST Code word. Traits are tested for with the SOF POP. (For further details, see "Traits", Section 1.13.1.)

4. If applicable, a diagram of the rungs of an entry, and their usage. Each of the positions within a list entry is called a rung, and is numbered beginning at zero. When a particular rung is referenced by number, it should be parameterized by haing a name that is used consistently throughout the compiler. As it happens, however, rungs are still sometimes referenced absolutely by number. Rungs are occasionally called by different names in different parts of the compiler. And not all rung names follow the convention that they should begin with the letter "Y".

   Sometimes a rung legitimately has two names. For example, the "used trigger" rung of the constant lists (e.g ICONSLST) is later used as the "literal location" rung, and therefore properly has two names. It is really serving two purposes, and this would enable the two to be separated, if such a thing were needed. In addition, some rung names are

used for more than one list. For example, the zero (constant value) rung of most of the constant lists is called YCONVAL.

5. A description of the list and its usage.

ALPHALST Alphanumeric List

This list is used to save up alphanumeric characters (one per word), when a part of the source string must be both syntactically scanned and also saved for inclusion in the object code. This is done by the FORMAT and OUTPUT statements. The saving takes place in Next Active Character. (See "Character Scanning", 2.2, and "Saving Alphanumeric Characters", 2.5)

ANDLIST AND List

A temporary acratch list used only in Logical Expression Scan (in Pass 1) to collect the operands of one or more .AND. operators.

ARGLIST Argument List

A temporary scratch list used in both Pass 1 and Pass 2 to collect and process arguments to functions or subroutines.

ARRAYLST Array List

Entry size - 3, 5, or 6 (actually defaulted to 1 in the Code word)

0	YAROFFST	Offset in elements (pointer to ICONSLST or expression)
1	YARSIZE	Size in elements (same)
2	YARDIMNS	Dimensions group (pointer to GROUPLST)
3	YARDUMES	Dummy messy expression (pointer to GROUPLST)*
4	YARDMPLX	Dummy plex list (pointer to GROUPLST)*
5	YARDPRIV	Dummy offset private temp (PTEMP pointer)**

* Present only on dummy arrays with adjustable dimensions

** Present only on dummy arrays with adjustable dimensions and non-constant offset

Contains additional information (that will not fit in the Symbol Table) about arrays, including pointers to the offset, the size of the array, and the group containing the subscript multipliers. For dummy arrays with adjustable dimensions, additional information is needed. The multipliers just mentioned are not simple integers, so they must be computed at run-time and stored in temps. The previous group therefore points to the temps. Another group contains the plex pointers for the messy (non-integer constant) multiplier expressions and the temps that these should be stored into (these correspond to the temps pointed to by the previous groups). Another group contains a complete copy of the Plex List needed to process the messy expressions (this is pointed to by rung 4.) Finally, if the offset is not an integer constant (it might be, even though the array has some adjustable dimensions), there needs to be a pointer to a private temp into which the offset (in words) is to be stored. (For further information, see "Arrays", 3.2, and "DIMENSION Statement", 3.36.)

ASNLBLST Assigned Label List

Entry size = 2 Key size = 1



Pointer to variable (typically to SYMTABLE)

Label of Label; i.e., created label for future code.

Pointer to CRELBLST.

This list is used by Pass 2 to help in the generation of those statements in which a variable name may optionally be used in place of a statement number. These statements include: Arithmetic IF, Computed GO TO, IF SENSE SWITHC, IF SENSE LIGHT, and IF OVERFLOW.

The assigned labels in these statements are essentially treated as Assigned GO TO statements to which the other statements can branch. All of the assigned GO TOs come out at the end of the statement. The Assigned Label List is used to register the labels that are used in any one statement. (It is cleared at the end of any statement.) By registering the labels, we can avoid some redundant code if the same label is repeated. For example, in the statement

IF (A+B) GRUNCH(J+K),23,GRUNCH(J+K)

the code for GO TO GRUNCH(J+K) is generated only once.

The items on the ASNLBLST are two words long, the first being a variable pointer (possibly subscripted) and the second being a created label pointer. The created label will later be attached to the point in the code where the code for this assigned label is. The only routines that use the ASNLBLST are Check For Assigned Label (CHEKLABL), Register Assigned Label (REGASLBL), and Generate Assigned Labels (ASLBLGEN). All of these are in Pass 2.

CHEKLABL does a rather peculiar thing if the assigned label happens to be the first one to be put on the ASNLBLST: it stores its created label in LABEL4. LABEL4 is a temp where statements such as Arithmetic IF store the label of the following statement, to see if they can generate code that "falls" into the next statement. When there are assigned labels, they will be generated between the IF statement and the actual next statement, so CHEKLABL puts the label of the first Assigned GO TO as the label of the next statement, and the IF branching logic falls into it.

ASTLIST Asterisk List

This list is used to interface the IBM form of non-standard return (using asterisks in the dummy list) with Xerox internal subprograms. For each FUNCTION and SUBROUTINE statement (there will be more than one only with internal subprograms), the ASTLIST contains the count of the maximum number of asterisks appearing in that statement's dummy list and the dummy lists of all succeding ENTRY statements, so that RETURNs can be diagnosed better than is needed. (See "FUNCTION Statement", 3.52, and "ENTRY Statement", 3.47.)

BLCOMLST Blank COMMON List

Entry size = 6 Key size = 2

Although the official entry and key sizes are as shown above, most of the time (i.e. during Pass 1) the list is used as if it had default entry and key size of 1. Each word is a pointer to a variable in the Symbol Table. The order corresponds to the order of blank COMMON; that is, the first variable in blank COMMON is at the top of the list, the last variable is at the bottom.

After this information has been processed by ALLOCATE, it is thrown away and a single six-word entry is placed on the BLCOMLST, so that it will look like the BLNAMLST (see below).

BLNAMLST Block Name List

Entry size = 6 Key size = 2

0	YBLNAME	Name of labeled COMMON block (first four characters)
1		(last four characters)
2	YBLSIZE	Block size (in words)
3	YBLPRTY	Block parity: 0 (no preference), 2 (even), or 1 (odd)
4	YBLPRTYC	Block parity culprit (pointer to SYMTABLE or zero)
5	YBLOVFL	Block overflow trigger (TRUE = exceeds maximum memory size)

For each labeled COMMON block used, Pass 1 creates this six-word entry, but sets up only the name. ALLOCATE sets up the rest, based on information in the LBCOMLST and the EQUIVLST. Furthermore, in order to process blank COMMON with the same code, the BLCOMLST, used by Pass 1 as a single word entry list, is set up by ALLOCATE with one (and only one) six word entry, the same as above with the name all blanks.

If the COMMON area contains any double precision, complex, or double complex variables, the parity culprit identifies the first such variable encountered, i.e. the one that froze the parity. The concept of parity no longer applies. An odd parity (COMMON block must begin at an odd location) is no longer legal because it depends on a non existent loader feature (see "Blank COMMON", Section 4.5.15).

CKCONLST Complex/Komplex Constant List

Entry size = 6 Key size = 4 Mode = complex

Traits: Addressable, Constant, Signed Addressable

0	YCONVAL	Real part (double precision value)
1		
2		Imaginary part (double precision value)
3		
4	YCCONUSD or YCCONLOC	Complex used trigger (TRUE = used as complex) -or- Complex literal location (relative to "P" origin)
5	YKCONUSD or YKCONLOC	Komplex used trigger (TRUE = used as komplex) -or- Komplex location (relative to "P" origin)

Pass 1 sets up all complex and double complex (komplex) constants in double complex form on this list. Based on the overall mode of the expressions in which these constants appear, Pass 2 then determines whether a single complex or double complex literal is needed (or both) and sets the corresponding "used trigger". Then in Pass 3, GENLIT generates the needed literals (if any) and puts their program locations back where the triggers used to be. Thus a single entry on the CKCONLST might generate zero, one, or two literals on the LITERL T (or it might generate two which overlap into one). (See also "Constants", 3.7, and "Generate Literals", 2.27.)

CLUMPLIST EQUIVALENCE Clump List

Entry size = 8 Key size = 1, although list is not searched

0	YHEADPTR	Head pointer (to SYMTABLE)
1	YTAILPTR	Tail pointer (to SYMTABLE)
2	YLOWELEM	Lowest element (numeric value)
3	YHIELEM	Highest element (numeric value)
4	YCLUMPTY	Clump parity (value = 0, 2, or 3)
5	YCLUMPTC	Clump parity culprit (pointer to SYMTABLE, or zero)
6	YDATACUL	DATA Culprit (pointer to SYMTABLE, or zero)
7	YCLUMOVF	Clump overflow trigger (.TRUE. or .FALSE.)

The CLUMPLST is used within ALLOCATE to carry most of the information about a set of variables (called a Clump) that have been EQUIVALENCEd to each other. The variables themselves are chained to each other by a thread that runs from the Head Pointer (YHEADPTR) through the Symbol Table to the EQUIVALENCE Chain List (EQVCHLST), back to the SYMTABLE, back to the EQVCHLST threading back and forth between those two lists until the last entry is reached.* This tail entry, in addition to being pointed to by YTAILPTR contains a zero in the Chain Pointer Rung (YCHAINPT) of the EQVCHLST; any other entry in the chain would have a SYMTABLE POINTER IN THIS RUNG, POINTING TO THE NEXT ENTRY.

YLOWELEM and YHIELEM are relative to the origin of the clump. The size of the clump (in allocated words) can be computed by subtracting YLOWELEM from YHIELEM. If this difference ever exceeds the maximum memory size, the Clump Overflow Trigger (YCLUMOVF) is set, but processing continues.

---

* A diagram illustrating the data structure of these lists appears under EQVCHLST.

If the Clump Parity (YCLUMPTY) is other than zero, then there will be a Clump Parity Culprit. That is, YCLUMPTC will be a pointer to the SYMTABLE to that DOUBLE PRECISION, COMPLEX, or DOUBLE COMPLEX variable that established a parity preference for this clump. This information is retained so that we can print a meaningful diagnostic in case parity rules are violated.

The other "culprit" rung is the DATA Culprit Rung (YDATACUL). If nonzero, this is a pointer to the SYMTABLE to the first variable encountered in this clump that has appeared in a DATA statement. This allows us to print

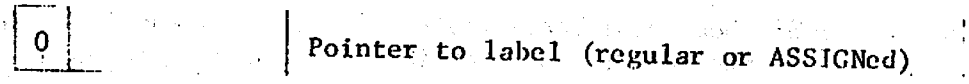
a meaningful diagnostic in case somebody ever tries to EQUIVALENCE this clump to a variable that is in blank COMMON.

For both Parity and DATA we keep track of only one culprit. If there are other culprits, we do not tell the user about them.

CODELIST Code List

Passes 1 and 2 use this list to accumulate their generated output before moving it to the OUTLIST. Pass 1 does this because the code may have to be thrown away if the statement fails, and because sometimes new code has to be inserted back before old code. Pass 2 also needs to go back and insert code (retroactive code generation), and furthermore has to do a first assembly pass on the code before outputting it. (See "CODELIST MANIPULATION IN Pass 1" and CODELIST Manipulation in Pass 2", 2.12 and 2.13.)

COMPGLST Computed GO TO List



This list is used rarely and only in Pass 2.

Pass 2 receives from Pass 1 a pointer to a Label Group Plex (LBLGRPLX) which contains all the labels. There may be two, there may be hundreds. Pass 2 needs a place to PUL that plex to so that it can work with the labels, and the COMPGLST is that place. See COMPGOSG, 3.31.2.

CPXTMLST Complex Term List

This a temporary scratch list used only by Pass 2. It is used in conjunction with the TERMLIST in generating sums and products. The sum and product generators separate the terms of a long sum or product into complex (i.e., COMPLEX or KOMPLEX) and non-complex. The terms are in effect sorted into these two categories regardless of the order in which the user wrote them, and then arithemtic for the non-complex terms is generated first. The CPXTMLST is used to aid in this sorting process.

CRELBLST Created Label List

Entry Size = 2 Key size = 0 (i.e., 1 but label field only) Mode=Label

Traits: Addressable, Signed Addressable

0	CRELBWD (YLABLNUM)	Created label value (as a number)
1	LBDEFWD (YLABELOC)	Label definition location (relative to P origin)

The CRELBLST is very similar in format to the OCLBLST (Local Label List) and the NLCLBLST (Non-Local Label List). It contains labels generated by the compiler. These labels appear on the object code listing as an integer number followed by the letter G.

The first rung of any label list entry (on any of the three lists) is the label number--i.e., the number that is printed before the S, S, or G on the object listing. For the S and S numbers, this is simply the number the user wrote. With the CRELBLST a rather clever technique is used to assure that these numbers will increase along with the program location counter. When a label is created (most are generated by Pass 2, but sometimes Pass 1 generates them), a zero is placed in the Label Value Rung, CRELBWD (note that this is a rung name that does not have an initial "Y". Then in Assemble Code (ASMCODE) in Pass 2, which takes place just before the code on theCODELIST is put out to the OUTLIST, a number is generated and stored into this rung. Since Assemble Code always assembles code in increasing order of location, the created label numbers also come out in this roder, rather than the order in which they were conceived.

DACONLST Data Constant List

Like the DVARLIST, the DACONLST is a temporary list used only in Pass 1 during the processing of the DATA statement. It is used much as the TERMLIST is used in Pass 2--it has a plex PULled to it, then the terms are TOLed one by one and processed. The terms in the case of the DACONLST are constant pointers and/or plex pointers. These plex pointers can be the simple 2-term Repeated Constant Plex kind (REPCONPX) or they can be the

multiple-term Constant List Plex kind (DCONLSPX), in which case we must reserve the DACONLST and process the terms individually.

DCODLIST Delayed Code List

The DCODLIST is used in Pass 2 to delay code. That is, code is generated at some time onto the Code List, moved over to the Delayed Code List, and kept there until some later time (usually several statements later) when it can be moved back onto the CODELISTand output through Assemble Code (ASMCODE).

The most typical example of the use of the DCODLIST is in the generation of loops. Here both the top and the bottom of the loop are generated, and the code for the bottom is delayed until later.

The EORCDLST (.EOR. Code List) performs a similar function, but only within a logical expression within one statement. Here the two paths of .EOR. code are generated simultaneously, but one path must follow the other in the object program.

DECLLIST Declaration List

Entry size = 2 Key size = 2

0	First half of name
1	Second half of name

This list is used in OUTBO in Pass 3 to keep track of the declarations that must be made in the binary object language. Eight-character names that correspond to the external references and definitions are kept on this list. There are also some artificial entries for control sections such as P (program location) and V (local variables) and blank COMMON. This list is searched to convert names to symbol numbers. Numbers are assigned by position on the DECLLIST.

DHEXLIST DATA Hexadecimal Constant List

Entry size = 5 Key size = 5

0	YDXVALU1	1st (left-most) hex value word
1	YDXVALU2	2nd hex value word
2	YDXVALU3	3rd hex value word

3	YDXVALU4	4th (right-most) hex value word
4	YDXSIZE	Size of hex constant in words

This list is used only in the processing of the DATA statement in Pass 1. By accident, however, the data on this list is never removed and remains in list memory throughout compilation.

The official entry and key sizes are both 5, but this is somewhat misleading in the case of Key Size, since the SER POP (Search) cannot handle keys bigger than four words. If this list were actually searched, it would need to be searched with five-word keys. But it is never searched. U NLIKE MOST CONSTANT LISTS, THE DHEXLIST does not get its constants registered on it; they are simply moved onto the list, and duplicates are not eliminated. This seemed a better solution than increasing the capacity of the SER POP.

DIMLIST Dimensions List

A temporary scratch list on which the dimensions (or subscript multipliers) of an array are collected or processed. The DIMENSION statement collects them there and puts them into a group on the GROUPLST, pointed to by the YARDIMNS word on the ARRAYLST entry. Other places, such as Combine Subscripts (in PASS0), Generate Array Offsets (in PASS2), and Next Equivalence Item (in ALLOCATE), pull that group back to the DIMLIST in order to operate on the dimensions.

DOLIST DO List

Entry size = 3 Key Size = 1

During Pass 1:

0	Terminal label (pointer to LOCLBLST or NLCLBLST)
1	Line count of DO or REPEAT statement (not pointer)
2	Control variable (pointer to SYMTABLE) for DO -or- zero value (not pointer) for REPEAT

The terminal label is used by Process Ends of DO-loops (in PASS1) to recognize the end of the loop. The line count is present so that error messages can identify the beginning of an unclosed loop. The DO-loop control variable pointer is used (1) by Outer DO Check (see "DO Statement", 3.38) to assure that no inner loop uses the same control variable as an outer one, and (2) by Combine Subscripts (see "Subscripts", 3.3) to see whether a given subscript expression is loop independent (i.e., is a simple expression involving only constants and DO-loop

control variables). These routines use SERK (Search with Key) specifying an offset of two in order to search for equality on the third word of DOLIST entries.

During Pass 2, DOLISTis used with an entirely different structure (though its official entry and key sizes remain unchanged) : each file on DOLISTis either empty, or it contains a four-word item describing how a DREGLIST item is to be restored at the end of a loop. The structure of this item is as follows:

)	Saved contents of the item's YDOCVAR rung
1	Saved contents of the item's YDOADND rung
2	Saved contents of the item's YDOLEVEL rung
3	Pointer to the item (DREGLIST pointer)

The meaning and use of DOLIST during Pass 2 is described in more detail in the section on DO-Reg Context Storage, 2.17.2.

DPAIRLST Data Pair List

The DPAIRLST is a scratch list of very limited use. It is used only in STASCAN in a routine called EXPDPRSQ (Expand DATA Pair Sequence) where it receives a file of DATA pairs from the WORKLIST. Then the pairs are taken off the top one at a time and processed. The items are logically two words long, although the item size is given the default value of one.

DPRSZLST Data Pair Size List

The DPRSZLST is kind of fun. It solves a fairly sticky problem in the DATA statement. The problem has to do with Failing during the expansion of a DATA PAIR. (A DATA pair is defined as a variable list followed by a constant list enclosed in slashes.) The rule about DATA statements is that if anything goes wrong with a DATA pair, the whole DATA pair is deleted. (The error message includes the sentence: DATA GROUP DELETED; in the reference manual we call DATA pair a DATA group.)

Now deleting a whold DATA pair can get fairly involved since a DATA pair can generate thousands and thousands of words. Remember, both the variable list and the constant list may be arbitrarily long.

Expand DATA Pair Sequence works by matching up variables with constants one variable and one constant at a time. If at any time a mode mismatch should occur (such as logical versus arithmetic), or an implied DO loop run amuck, or a subscript get out of range, the whole DATA pair is to be scrapped.

One way of handling this problem would be for PASS 1 to save up all its output for a particular DATA pair until the whole pair was expanded, and then perform the output if the pair went bad. This would place a restriction on how much data a particular DATA pair could produce (and a fairly stiff restriction on small-memory machines; remember: this compiler was originally intended to be able to run in a 16K memory machine, and it still can with a small monitor.) Therefore, we output the words of data to Pass 2 every time a variable is matched with a constant; there are about five words each time.

Now we come to the Data Pair Size List. Pass 2 will receive the output for a DATA pair, even if the pair later turns out defective. Through the wonders of the DPRSZLST, it can ignore these words. As Pass 1 expands a DATA pair, it keeps track of how many words it generates. If a failure occurs, Pass 1 puts the word count on the bottom of the Data Pair Size list. If the pair gets through without error, a zero is appended. The words on the DPRSZLST essentially tell Pass 2, "Ignore this many words from the input stream". DPAIRGEN (DATA Pair Gen) in Pass 2 performs this ignoring function. Pass 1 has sent a Data Pair Number (DPAIRNO) indicating that this should be done, and upon receiving it Pass 2 takes the top word from the Data Pair Size List and ignores that many words from the Input List.

DREGLIST DO Register List Static)

Entry Size = 7 Key Size = 1

0	YDOCVAR	DO-Register Control Variable (of loop)
1	YDOADND	Addend
2	YDOCLOB	Clobbered Trigger
3	YDONR	DO-Register Number (as integer 2, 4, or 6)
4	YDOX	DO-Register Number in X-Field
5	YDOR	DO-Register Number in R-Field
6	YDOLEVEL	DO Level Pointer (to DOLIST)

This is a static list of exactly three items--one for each possible DO register. On the Sigma, the DO registers are the even-numbered index registers, i.e., 2, 4, and 6.

This list is used only in Pass 2 whree it keeps track of the contents of certain registers that are dedicated to DO loops. See 2.17.2 for the details.

The actual contents of this list are in the assembly called FORTTEMP.

DREPLIST DATA Constant Repeat Count List

The DREPLIST is a push-down list used only during Expand DATA Pair Sequence in Pass 1. It is used to unnest the structures like 3*(4.7,-13.82,.TRUE.,(16,-1)). Here the repeat count is 3, and this is appended to the bottom of the DREPLIST. This bottom is decremented as the constants are repeatedly generated. We need a list for this rather than a plain temp because repeated constants can be nested, as in the statement

```
DATA GLOP /5*(14,37*(42.8E6,2*1,16))/
```

in which three separate (and nested) repeats are going on.

DSUBLIST DATA Subscript List

Entry size = 3 Key size = 1 Mode = Integer

Traits: Addressable, Signed Addressable

A temporary list used only in Pass 1 for DATA statements. It is used both during scanning of the statement and during expanding of DATA pairs, and its format differs considerably, though in both cases the entry and key size are the same.

During scanning of DATA statements:

0		Subscript variable (pointer to Symbol Table)
1	DCHCTWD	Character count of last character in variable name
2	DOUSEDWD	Used as DO control trigger (TRUE = used)

This usage of the DSUBLIST serves only to give excellent diagnostics on DATA statements with implied DO loops. It facilitates two checks. First, any variable that appears in a subscript (or an inner DO-control expression) must show up as the control variable of an outer DO. After each nest of DOs is scanned, any variable in this list that does not have the "used" trigger set is an error, and the character count tells us where to mark the error. Second, every DO-control variable should be used in a subscript (or DO-control expression) in an inner DO. Otherwise it is wasted and is not likely to be what the user intended. (See "DATA Statmenet", 3.34.)

During expanding of DATA pairs:

0	YVALUEWD (YVALUE)	Current value of DO index
1	YSTEPWRD	Value of step (increment)

2	YCNTWORD	Total DO iteration count

DUMMYLST Dummy List

A temporary scratch list used only in Pass 1 to collect and process the dummies of FUNCTION, SUBROUTINE, and ENTRY statements. Statement functions do not use this list. They pile up their dummies on the WORKLIST.

DVARLIST Data Variable List

Like the DACONLST, the DVARLIST is a temporary list used only in Pass 1 during the processing of the DATA statement. It is used much as the TERMLIST is used in Pass 2--it has a plex PULled to it; then the terms are taken off the top one by one and processed. The terms in the case of the DVARLIST are variable pointers and/or plex pointers. These plex pointers can be of the Data DO Control kind (DVLWDPX) which consist of six terms, or they can be the Variable List Plex kind (DVARLSPX), in which case we must reserve the DVARLIST and process the terms individually.

EASYLIST Easy Term List

The EASYLIST and the HARDLIST are temporary scratch lists used in Pass 2 to separate terms into two classes. This is chiefly in the generation of the .EOR. operator, for which hard terms and easy terms are handled in quite different ways.

These two lists function rather analogously to the way the TERMLIST and the CPXTMLST function during the generation of sums and products.

ENTRYLST Entry Point Code List

This list used to hold code generated for FDP (FORTRAN Debug Package--the run-time debug package) in connection with entry points to the generated code (see 3.19). As defined through the parameter YYENTRY, the item size for this list is five, although it is given the default value of one.

One reason for generating this code on the ENTRYLST instead of the CODELIST is so that the ENTRYLIST can be counted to see how many items were generated. (The CNT POP is used, followed by a DVW by YYENTRY, i.e., 5, to see how many entry points there were.)

In Pass 2, in DEBUG mode, there will be one entry on this list for every SUBROUTINE, FUNCTION, or ENTRY statement in the program. Those for ENTRY statements have the Indirect bit of the first word set; those for SUBROUTINEs and FUNCTIONs do not. Statement functions (ASFDs) do not get entered on this table at all.

Code is accumulated on the ENTRYLST from the compilation of SUBROUTINE, FUNCTION, and ENTRY statements. Then it is all put out together at the end among the debug tables (see 3.19.2). See also LABELLST and SRCLNLST.

EORCOLST .EOR. Code List

In Pass 2, for the generation of the .EOR. logical operator, it is sometimes desirable to generate two parallel and nearly identical paths of code. In the final object program, one path of code follows the other, but it is handy to generate them both at once. So the EORCOLST is used as a kind of Delayed Code List for the second path.

EORLIST EOR List

A temporary scratch list used only in Logical Expression Scan (in Pass 1) to collect the operands of one or more .EOR. operators.

EQUIVLST Equivalence List

Entry size = variable (defaulted to 1 in Code word)

0	EQSETBIT + source line number of equivalence set (group)
1A	First variable in equivalence group (pointer to Symbol Table)
2A	Subscript count (never zero; 2A-nA absent if no subscripts)
3A	First (leftmost) subscript value (not pointer)
.	.
.	.
.	.
nA	Last (rightmost) subscript value
1B	Second variable in equivalence group
.	.
.	.
.	.
1N	Nth (last) variable in equivalence group

EQSETBIT is misnamed. On the 9300, it was actually a bit, but here it is a pointer number (like a list or a plex) in the upper eight bits. The source line number is used to give good diagnostics in ALLOCATE, after it is too late to mark positions on the source image. The EQUIVLST rungs have no names, since the entry size is variable. In addition, ALLOCATE flips the list before processing, so that it can take the items off the bottom instead of the top. Thus everything is backwards from the way it was put on.

The following shows the eleven words that would be put on the EQUIVLST by the EQUIVALENCE statement processor for the statement.

```
EQUIVALENCE (P,Q(3),R( -4)), (X,Y)
```

1. EOSETBIT + current line number
2. Pointer to P
3. Pointer to Q
4. 1
5. 3

6. Pointer to R

7. 1

8. -4

9. EQSETBIT + current line number

10. Pointer to X

11. Pointer to Y

Note that at the time these values are placed on the EQUIVLST, it is not necessarily known whether the "subscripts" are actually subscripts or are position counts, or whether there are a legal number of them. This is determined later on during allocation (see ALLOCATE, Section 3.8).

EQVCHLST EQUIVALENCE Chain List

Entry size = 2 Key size = 1, although this list is not searched.

0	YCLUMPTR	Clump pointer (to CLUMPLST)
1	YCHAINPT	Chain pointer (to SYMTABLE, or zero if end of chain)

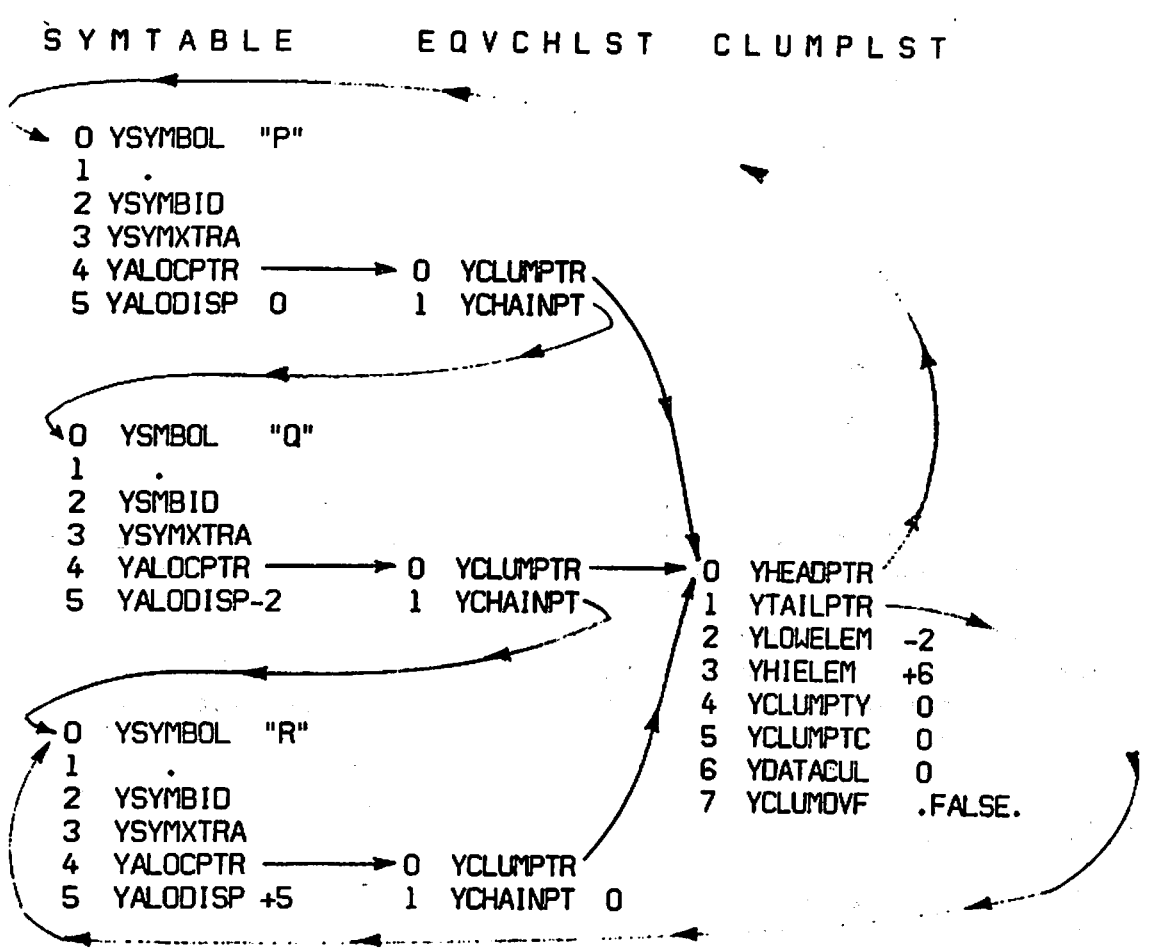
The only reason that the EQVCHLST exists at all is that ALLOCATE needed another word in the SYMTABLE, and it did not seem worthwhile to add one for such a limited purpose. If there were an extra word on the SYMTABLE, then Rung 4 (YALOCPTR) would point to the next SYMTABLE item in the chain, and Rung 6 (the new rung which we would call YCLUMPTR) would point to the CLUMPLST. Rung 4 would be serving the function of YCHAINPT.

The problem that the EQVCHLST solves is that, for every Symbol Table entry, we need a pointer to tell what clump it is in, if any, as well as a pointer to the next item in the chain. Three lists are involved in the representation of an EQUIVALENCE clump, and the following diagram may help you understand how the three tables are intertwined. The data shown is for the EQUIVALENCE of three variables. This involves three items on the Symbol Table, three items on the EQUIVALENCE Chain List, and one item on the EQUIVALENCE Clump List. The statement represented is

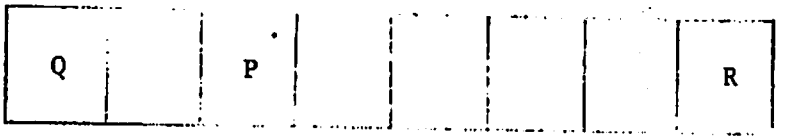
EQUIVALENCE (P, Q(3), R(-4))

where P, Q, and R are scalars of one word apiece. If they were arrays whose lowest subscript bound was 1, the only possible change in this data would be in YHIELEM; lower bounds other than 1 might alter YLOWELEM, but the point is that this data structure would be true for scalars or arrays.

Data Structure for: EQUIVALENCE (P, Q(3), R(-4))



By the way, the EQUIVALENCE statement implies the following word allocation:



ERRORLST Error List

Entry size = 2 Key size = 1

0		Character count of error position
1		Location of error message (key and text)

Used during scanning in Pass 1 to pile up possible (delayed) error messages. Later, if appropriate, these messages are sorted (by character position) and all printed at once following the statement. The ERC, ERL, and ERW POPs are used to put entries on this list. (See "Diagnostics" in Chapter 2.)

EXITLIST Exit List

Maintains the stack of exit locations, so that the compiler can operate recursively. Exits are typically added to the list by the JRS POP (Jump Recursively to Subroutine). The actual address of the JRS is placed on the list; thus, return is (via a BIR) to the following location. Each entry in the EXITLIST also carries (in the sign bit) the answer flag for its recursive level (TRUE = sign bit set). For the zero level, Initialize Lists places a dummy word on the exit list, so that there is a place for theanswer flag.

GLNAMLST Global Name List

Entry size = 5 Key size = 2

0	YGLNAME	Name of global variable (first four characters)
1		(last four characters)
2	YGLSIZE	Size (in words)
3	YGLPRTY	Parity: 0 (no preference), 2 (even), or 3 (odd)
4	YGLPRTYC	Parity culprit (pointer to SYMTABLE)

This list is used in essentially the same way as the BLNAMLST (Block Name List), described previously. It is set up by ALLOCATE, rather than by Pass 1. There is no overflow trigger rung, because a global variable cannot be extended, and if the variable itself is dimensioned too large, Pass 1 will detect that and disallow it.

GROUPLST Group List

The Group List might legitimately have been called the Permanent Plex List. It was created because, although most plex information can be flushed away at the end of statements, certain plex information is needed later. This plex information is chiefly that generated for array declarations, especially those arrays that have adjustable dimensions.

The items on the Group List begin with a count word n, then n words of information. Groups are created by the COG (Copy Group) POP. One writes

```
COG   SOMELIST
```

and the COG POP takes the bottom file of that list and makes it into a group on the bottom of the GROUPLIST. (It manufactures the count word automatically.) It empties the bottom file of SOMELIST, and appends a pointer to the group onto the bottom of the WORKLIST.

This pointer points at the count word* and is just what is needed for the PUG (Pull Group) POP. This POP expects a GROUPLST pointer in W0 and pulls the contents of the group (however many words there may be, including zero) to the list addressed. It also removes the pointer from W0.

Once data has been entered on the GROUPLST, it is never removed, for it may be needed throughout the whole compilation.

---

* The pointer contains the number of the GROUPLST in its ID field. This differs from the convention for plex pointers.

HARDLIST Hard Term List

The HARDLIST and the EASYLIST are temporary scratch lists used in Pass 2 to separate terms into two classes. This is chiefly in the case of the .EOR. logical operator, for which difficult terms and easy terms are handled in quite different ways.

These two lists function rather analogously to the way the TERMLIST and CPXTMLST function during the generation of sums and products.

HCONSLST Hollerith Constant List

Entry size = 2 Key size = 1 Mode = Integer

Traits: Addressable, Constant, Signed Addressable, Test Addressable

0	YCONVAL	Value of constant (either left or right-justified)
1	YIHXUSED or	Constant used trigger (TRUE = used) -or-
	YIHXLOC	Constant literal location (relative to "P" origin)

All H and R form Hollerith constants are kept on this list. They could be kept on the ICONSLST, since they are integer, but this tells the object listing to show them in alphaumeric. If the constant is actually used in the generated code, Pass 2 sets the "used" trigger. In this case, GENLIT puts the constant value on the LITERLST (to become a generated literal) and puts the program location of the literal back on the HCONSLST where the trigger used to be. (See also "Constants", 3.7, and "Generate Literals", 2.27.)

ICONSLST Integer Constant List

Entry size = 2 Key size = 1 Mode = Integer (but see below)

Traits: Addressable, Constant, Signed Addressable, Test Addressable

0	YICONVAL	Value of constant
1	YIHXUSED or	Constant used trigger (TRUE = used) -or-
	YIHXLOC	Constant literal location (relative to "P" origin)

All integer and logical constants are kept on this list. The logical constants are differentiated only by the mode of the pointers to them. When a logical constant is registered on this list, the pointer is changed to logical mode, so that the object listing can show the constant as

.TRUE. or .FALSE., even though the constant is on the Integer Constant List and actually has the value -1 or 0.

If the constant is actually used in the generated code (as either logical or integer), Pass 2 sets the "used" trigger. In this case, GENLIT puts the constant value on the LITERLST (to become a generated literal) and puts the program location of the literal back on the ICONSLST where the trigger used to be. (See also "Constants", 3.7, and "Generate Literals", 2.27.)

IFLIST IF List

For each arithmetic (or device) IF statement, Pass 1 places on the IFLIST the label (statement number) of the following statement, so that Pass 2 can compare that label with the possible transfer labels of the IF statement and determine whether it can fall into one of them. Pass 2 can then reorder the testing (test for the other conditions first) and not have to generate a branch to a location that follows immediately. If the statement following the IF statement does not have a label (and is not a declarative statement), an absolute zero is placed on the IFLIST instead. The routine Check Label After IF sets up the IFLIST, based on the AFTERIF trigger that is set by the non-logical IF statements. (See "Arithmetic IF Statement", 3.55.1.1 and 3.55.2.1.)

INDOLIST Inner DO List

Entry size = 1 Key size = 0 (i.e., but label field only)

This is used only in Pass 1 during scanning of DATA statements. For each implied DO loop in the variable list, a pointer to the control variable is placed on the INDOLIST. Each DO loop also therefore searches this list to make sure that no inner loop uses the same control variable; if the control variable already appears on the list, an error is given. After each nest of loops, the INDOLIST is ZAPed to begin again.

The key size is 0 (usually used for labels, which need to search on the lower twenty bits and ignore the label flags in the upper bits) in order to effect a label-type search. The Symbol Table pointer for the control variable may be different by the time it appears again, by virtue of having also appeared as a regular DATA variable and therefore acquiring a class, which will change the upper bits. Restricting the search to the lower bits assures that a change in class will not be noticed, and there is no problem with doing this, because the only thing that appears on the INDOLIST are pointers to the Symbol Table.

INPUTLST Input List

Used by all three passes to obtain their input. Pass 1 reads source cards and appends them to this list. The other two passes use, as their input, the output from the previous pass. To accomplish this, the actual list parameters (Base, Start, etc.) for the OUTLIST are moved to the INPUTLST. Thus, in effect, the OUTLIST becomes the INPUTLST. Note also that if the output from one pass overflows to a scratch file, the next pass will have to keep reading from that file to fill up the INPUTLST.

There are a number of ways in which the INPUTLST is handled specially. See "Initialize Lists" (Section 1.4.8). "Reassign Memory" (Section 1.4.9), "Pass 1/Pass 2 Interface" (Section 2.11), and "Character Scanning" (Section 2.2).

INTRILST Intrinsic List Static

Entry size = 3 Key size = 2

0		Name of intrinsic function (first four characters)
1		(last four characters)
2	YTYPEACL	Type and argument class

This is a static list in the assembly EXSCAN, which contains the names and attributes of all the intrinsic functions recognized by the compiler. Rung 2 has the function result type in the usual mode field (third hex digit), and has, in the address field, the location to which Expression Scan should transfer to process the function's arguments. There are six classes of argument processing:

F1: 1 argument, Integer, Real, or Double precision (e.g. ALOG)

F2: 2 arguments, I, R, or D (e.g. AMOD)

F3: 1 or 2 arguments, I, R, or D (e.g. ATAN)

F4: any number of arguments, I, R, or D (e.g. AMAX)

F5: 1 argument, I, R, D, C, or K (e.g. AIMAG)

F6: 1, 2, or 3 arguments, I, R, or D (e.g. IF)

When an intrinsic function is identified, this class word is stored in the YINTRCL word of the Symbol Table entry. Then, since the address field contains an actual location, the next time the function is encountered, its arguments can be processed by branching indirectly through the YINTRCL word of the Symbol Table.

There is one additional attribute associated with an intrinsic function, and that is which one it is. This is determined by its position within

the INTRILST. If it is the 14th function, say, Pass 1 stores 14 in the YINTRNR word of the Symbol Table entry, so that Pass 2 can identify the function by number rather than by name.

The actual contents of the INTRILIST are in the assembly EXSCAN.

KLUDGLST Kludge List

Entry size = 2 (defaulted to 1 in Code word)

0		Pointer to clobbered symbol (in SYMTABLE)
1		First four characters of the name

Statement function definitions (and also FUNCTIONs) need to be able to create a new Symbol Table entry with the same name as an old one.

During some temporary period, we must assure that the new one will be found by searches rather than the old one, despite the fact that the old one will be encountered first in the Symbol Table. To accomplish this, the old one is temporarily clobbered, by storing all ones in the first half of the name. The original first half of the name is saved on the KLUDGLST, along with a pointer to its SYMTABLE entry, so that it can later be restored. (See "Statement Function Definition", Section 3.76.1.2.)

LABELLST Label Table Code List

The name LABELLST makes this list sound more important than it really is. It is used only in Pass 2 in the DEBUG (FOP) mode. It holds code that is generated for each label at definition time. Two words appear on this list for every label, although the item size has the default value of one.

Near the end of compilation in Pass 2, this code goes out all together as part of the Debug Tables (See 3.19.2). See also ENTRYLST and SRCLNLST.

LBCOMLST Labeled COMMON List

Entry size = variable (defaulted to 1 in Code word)

0		Labeled COMMON block (pointer to BLNAMLST)
1		First variable specified (pointer to SYMTABLE)

.	
.	
.	
N	Nth (last) variable specified

This list is set up by the COMMON statement in Pass 1 and is processed and emptied by ALLOCATE. The information contained here is distributed into the SYMTABLE (each symbol specifies that it is in a certain position of a certain COMMON block) and the BLNAMLST (name and size of each block).

Note that one entry on this list does not necessarily define a block completely. It may be reopened in another COMMON statement (or even in the same one). ALLOCATE has to collect all the LBCOMLST entries that point to each block name.

LCONSLST Label Constant List

Entry Size = 2 Key size = 1

0		Label pointer, to LOCLBLST or NLCLBLST
1	LBLOFLBL	Label of label--pointer to CRELBLST

This list is used in Pass 2 in conjunction with statement labels that have appeared in ASSIGN statements or as arguments to subprograms. These labels are generated as constants, which consist of B (Branch) instructions to the labels.

Labels are registered on this list so that only one branch will be generated for each label, no matter how many times it is used. Each label constant has a created label attached to it, and this is called "Label of Label".

Labels are registered by Register Label Constant (REGLBCON), and the constants are generated near the end of Pass 2 by Output Label Constants (OLBLCON).

LITERLST Literal List

Entry size = 1 or 2 (defaulted to 1) Key size = 1, 2, or 4 (defaulted to 1)

This list is created by Pass 3 (in GENLIT) and contains the literals (constants) that will be generated at the end of the object program. These literals are taken from the various constant lists, depending on how and whether the constants are actually used (see "Generate Literals", Section 2.27). The constants are registered on the LITERLST with an overlapped search that depends on the size of the constant. A one-word constant (e.g. integer) uses the default values of entry and key, namely, 1 and 1. Two-word constants (double and complex) use 2 and 2. The four-word constants (double complex) are rather peculiar. The key size must be 4, of course (the search must find a four-word match), but the entry size can be 2. A double complex constant might overlap the imaginary part of one previous literal and the real part of the following one. This is the only place where a search is done with a key size bigger than the entry size. Since these sizes are defaulted to 1 in the Code word for the LITERLST, the search is done with a SERK (Search with Key).

LOCLBLST Local Label List

Entry size = 2 Key size = 0 (i.e. 1 but label field only) Mode = Label

Traits: Addressable, Signed Addressable

0	YSULWORD (YSOLWORD) and YLABLNUM (YLABELNO)	SUL/SOL traits and label value (as a number)
1	YLABELOC (LBDEFWD)	Label definition location (relative to "P" origin)

All local labels (i.e. ordinary labels, not followed by $) are kept on this list. The second word contains the program location at which the label is defined. The first word contains the label value in the lower twenty bits (label field) and the SOL/SUL traits in the upper 12 bits. These traits are:

DEFINED	= X'80000000'	Defined
DEFEXEC	= X'40000000'	Defined on non-FORMAT statement (usually executable)
DEFMULT	= X'20000000'	Defined more than once (trait no longer used)
REFRNCED	= X'10000000'	Referenced (trait set up but never tested)
REFFORM	= X'08000000'	Referenced as FORMAT statement
DOLABEL	= X'04000000'	Label in DO or REPEAT range

ACTIVLBL = X'02000000' Active (possibly branched to)

This list probably has more different, poorly coordinated, names for its rungs than any other. These could be consolidated to just YSULWORD, YLABELNUM, and YLABELOC.

MODELIST Mode List

The Mode List is a push-down list used in Pass 2 to keep track of what mode a particular arithmetic expression is being generated in. The reason that it has to be a push-down list is that subprogram arguments have to be evaluated in their own mode, which may differ from that of the whole expression. Also, the right-hand operand of the ** operator (exponentiation) is evaluated in its own mode if it be integer.

Mode is set by BRLing to a little machine-language subroutine called Mode Set ML (MODSETML). This could also be accomplished by

```
MOK   MODELIST
```

for it appends the mode of the bottom of the WORKLIST to the bottom of the MODELIST. The reason it was not done with the MOK POP is that there was no MOK POP on the 9300.

The usual ways of getting rid of a mode are

```
MMO  MODEBOT
```

or by exiting through a machine-language routine

```
BRL  MODREXML  Mode Release and Exit ML.
```

Only the Mode Field (bits 8-11) are important in any entry on the MODELIST.

NAMELST NAME LIST List

Contains pointers to all the variables that are to be output as part of the NAME LIST. These are usually specified by the NAME LIST statement, but in run-time DEBUG mode, all appropriate variables are put on the NAME LIST. In order to assure that searches will succeed, despite changes in class that have occurred since the name was registered (e.g. a symbol is unknown class when registered, but is known to be a scalar when the search takes place), each pointer is declassified before being registered. That is, the ID field (first eight bits) simply points to the Symbol Table, rather than saying SCALAR, or ARRAY, or whatever.

NLCLBLST Non-local Label List

Entry size = 2 Key size = 0 (i.e. 1 but label field only) Mode = Label

Traits: Addressable, Signed Addressable

This list is exactly the same as the LOCLBLST (Local Label List), except that it contains the global labels (i.e. followed by S). The rung names, traits, etc. are the same.

OPCODLST Op Code List (Static)

Entry size = 4 Key size = 2

0		Name of op-code (first four characters)
1		(last four characters, e.g. for BIOANR or LLOCKS)
2	YOPCODE	Op-code value (and some structural characteristics)
3	YOPCLASS	Op-code class (traits pertaining to allowable usage)

This is a static list contained in SINCOLM1 that identifies all of the recognized op-codes for in-line assembly (S in column 1) and gives information about how each op-code may be used (in the source program) and how it should be constructed. The YOPCODE word has information merged into it and is then filed as part of the interface information for later passes. The first eight bits therefore contain a special, pointer-like, Pass 2/3 interface code. The YOPCLASS word uses only the first nine bits, for "permissibility" bits, indicating what kind of register field, address field, etc. may be used with this op-code.

The actual contents of this list are in the assembly SINCOLM1.

OPCSLIST Calling Sequence Op-code List (Static)

Entry size = 2 Key size = 1

0		Op-code value (right-justified)
1	YOPCSMNE	Op-code mnemonic (four characters)

This static list in OUTPROG identifies the calling and receiving sequence op-codes that are printed on the object listing. They must be kept separate from the rest of the op-codes, because they use some of the same values (e.g. SNGL looks like LCFI). The reason that this is a list, while the ordinary op-codes are kept in a table, is that the ordinary op-code table has a position allocated for every possible op-code value, while this has only a small number of possible values, which must be searched

for. Searching is much easier if the table is called a list and thus has parameters for the SER POP to use.

The actual contents of this list are in the assembly called OUTPROG.

ORLIST OR List

A temporary scratch list used only in Logical Expression Scan (in Pass 1) to collect the operands of one or more .OR. operators.

OUTLIST Output List

Used by Pass 1 and Pass 2 to file intermediate output for the following pass. This list is expected to grow very large, so if working storage runs out, the OUTLIST can be dumped onto an external scratch file, and later read in piece by piece and added to the INPUTLST. Most of the information destined for the OUTLIST is accumulated first on the CODELIST. (See "CODELIST Manipulation in Pass 1," Section 2.12, and "CODELIST Manipulation in Pass 2", Section 2.13.) Obviously, the OUTLIST must be handled specially during list reassignment, since it is expected to grow continously. (See "Initialize Lists", Section 1.4.8, and "Reassign Memory", Section 1.4.9.)

PARAMLST Parameter List

This is a general working list that usually carries parameters or flags that will be in use by a section of code. It can bee used in any way that is convenient (Expand DATA Pair Sequence actually uses it as an alternate Exit List), but typically an address is placed in the bottom word with an ADR POP and subsequent code references indirectly through P0. (P0 is a specially defined symbol referring to the bottom word on the PARAMLST. POIND, or *P0 is used to reference indirectly through it.) (See "A Few Basic Lists", Section 1.4.6, and "Special Addressing", Section 1.6.2.)

PLEXLIST Plex List

The Plex List is one of the cornerstones of the compiler. It contains variable-length data structures of a fairly complicated form that are used to represent expressions, as well as other things. The section on Plexes (1.10) elucidates these structures.

Data can get onto the Plex List by means of the Plex-Building POPs. In Pass 2 and Pass 3, data usually gets to the PLEXLIST from the INPUTLST or GROUPLST; these are plexes that were originally generated by other passes and were passed along as part of the input stream to the next pass.

In certain cases (see "Plex Passing" 3.15), Pass 1 elects to send only part of the Plex List to Pass 2.

Each plex begins with a count word telling how many words are in the plex (including the count word itself). The plex-building POPs employ a variable-length search to see that two identical plexes are never generated. This assures that pointers to identical expressions will be identical and thus easy to recognize.

PRODLIST Product List

A temporary scratch list used only in Expression Scan (in Pass 1) to collect the operands of one or more "*" (multiply) operators.

PXEMBLST Plex Embryo List

The Plex Embryo List is used only within the Interpreter (INTERPTR). It aids in the work of the FIP and FIC POPs. You see, FIP and FIC FIC are just like CIC and CIF except that they take a fixed number of terms from the WORKLIST instead of taking a variable number of terms from some other list. To save code in the INTERPTR, FIP and FIC take their fixed number of terms off the WORKLIST and place them on the Plex Embryo List, then address the Plex Embryo List and fall into common code for CIC or CIF. Thus the only time there is any data on the Plex Embryo List is during the actual execution of a FIP or FIC POP. Between POSs, this list is always empty.

RDCONLST Real/Double Constant List

Entry size = 4 Key Size = 2 Mode = Real (but see below)

Traits: Addressable, Constant, Signed Addressable, Test Addressable

0	YCONVAL	Constant Value (first word of double precision value)
1		(second word of double precision value)
2	YRCONUSD or YRCONLOC	Real used trigger (TRUE = used as real) -or- Real literal location (relative to "P" origin)
3	YDCONUSD or YDCONLOC	Double used trigger (TRUE = used as Double) -or- Double literal location (relative to "P" origin)

Pass 1 sets up all real and double precision constants on this list in double precision form. Based on the overall mode of the expressions in which these constants appear (and the ADP trigger, if requested), Pass 2 then determines whether a single or double precision literal is needed (or both) and sets the corresponding "used" triggers. Then in Pass 3, GENLIT generates the needed literals (if any) and puts their program locations back where the triggers used to be. Thus a single entry on the RDCONLST might generate zero, one, or two literals on the LITERLST (or it might generate two that overlap into one). If a single precision (real) literal is needed, the double precision value must be converted to single precision, complete with rounding. See also "Constants" (Section 3.7) and "Generate Literals" (Section 2.27).

REFLIST External Reference List

Entry size = 2 Key size = 2

0	YNAME or YSYMBOL	First four characters of name
1		Last four characters of name

This list keeps track of the library routine and DCB names that must be referenced to make the object program work. It does not contain the names of FORTRAN subprograms that are EXTERNAL (these are put out by OUTBO by a special sweep through the Symbol Table), but it does get every other name that must be REFed.

Most of the names that wind up on this list are links contained in P2P3ROOT, and in an early version of the compiler this list had single-word entries that were the addresses of the names. This seemed nice and economical--why use two words instead of one? Then along came SINCOLM1 (S In Column 1--in-line machine code) that can call out any name it wishes, including those that are not in P2P3ROOT. So the mechanism had to be changed to include the text of the name instead of its address.

Aside from SINCOLM1, all the names get on the REFLIST during Pass 2, and it all happens in a subroutine called RGREFNAM (Register REF Name).

The most obvious place where this routine is called is ASMBAL6 (Assemble BAL,6 Instruction) in Assemble Code. This processes things generated by the BLC (Build Library Call) POP. Oddly enough, there are eight other calls on RGREFNAM in Pass 2. If we are compiling a main program, we reference F:108, M:DO and M:OC. If we are compiling a main program under the DEBUG option, we additionally reference F:UF and M:SI.

Then there is CHEKUNIT that handles I/O units. It checks for BCD Read/Write statements that reference units 105 or 108, respectively, and register the names F:105 and F:108 if necessary. If a BDC I/O statement references a non-constant unit, then we reference several units, just in case it should turn out to be them. In particular, for a Read statement we reference F:101, F:103, and F:105; for a Write statement we reference F:102, F:104, F:106 and F:108. This makes sure that these file names have the appropriate characteristics at run-time.

SINCOM1 and Pass 2 have created the data on the REFLIST, and it remains for various parts of Pass 3 to use it. The assembly PASS3 uses it to print REF lines as part of the Object Listing (LO). OUTPROG (Output Program) SOTs for REFCODE to print such a name on the object listing. OUTBO (Output Binary Output) makes a sweep through the REFLIST to output any such names as REFs. And finally, SUMMARY (Print Summary) sorts the REFLIST into alphabetical order and prints the names under an appropriate heading.

REPLIST Repeat List

This is a temporary scratch list used in Pass 2 to process the REPEAT FOR statement. After the "=" sign in the REPEAT FOR statement comes a list of expressions and/or REPEAT Triples. These are sent to Pass 2 as one item--a Repeat List Plex (REPLSPLX) which gets PULled to the REPLIST from which the items are TOTed one at a time. A special case is made if the list contains only one item. In this case (a very common case), the contents of the loop need not be made into a subroutine and called with various values of the control variable. The REPLIST is left empty at the end of every statement.

REPLLIST Replacement List

A temporary scratch list used in Pass 1 and Pass 2 to collect and process replacements (i.e. assignment statements or embedded assignments). This needs to be a list because a replacement may have several left-hand (stored into) parts. For example,

```
X = Y = A(I,J) = B + SQRT(ALPHA)
```

SAVELIST Save List

Entry size = variable (officially 2 in Code word) Key size = 1

The SAVELIST is used in a variety of different ways. In Pass 1 it is used in the TRY-FAIL mechanism to save list parameters and single values. Single values are saved with a two-word entry, the first being the value and the second its location. Lists are saved with an entry of from one to four words, depending on how many of the list parameters are not equal to one another. A saved list entry always has the sign bit set in the bottom word, whereas a single value never does. The first three hex digits of the bottom word determine how many other words are present in the entry. Each of those digits that is all ones (X'F') specifies that one additional word is present in the entry. The meaning of these words is explained in "Saving and Restoring Information", Section 2.7.1.

In Pass 2, the SAVELIST IS USED IN TWO WAYS. Sometimes individual items are saved (with the SAV and RST POPs). This produces only a one-word entry containing the value. There is no location; it is the user's responsibility to remember the locations he saved and restore them to the same locations. Elsewhere, the SAVELIST is used just as a scratch list, to dump files of information from other lists, so that it can be copied back later.

Finally, Retroactive Subscript Gen (in Pass 2) uses the SAVELIST to keep track of index registers that need to be saved and restored in the generated object code. This is the only place that uses the official entry and key sizes (2 and 1). Each entry has this form:

0		XREG pointer (pointer to XREGLIST)
1		Temp pointer (fake list pointer)

SCONSLST String Constant List

Entry size = variable (defaulted to 1 in Code word) Mode = Unknown

Traits: Addressable, Constant

0	YSCONCNT (YSCONSZC)	Character count
1	YSCONUSD or YSCONLOC	Constant used trigger (TRUE = used) -or- Constant literal location (relative to "P" origin)
2	YSCONVAL	First word of constant value (first 4 characters)
•		
•		
•		

N		Last (N-1st) word of value (last 1-4 characters)

All alphanumeric string constants of more than four characters are kept on this list. Strings of four or fewer characters are treated as Hollerith constants (of integer type) and are kept on the HCONSLST. String constants specifically have no type. They are passed in calling sequences with a PZE, as opposed to INTG. Because of the variable length entry, the SCONSLST cannot be searched with the SER (or SERK) POP. Constants have to be registered with a special machine language search.

If the constant is actually used in the generated code, Pass 2 sets the "used" trigger. In this case, GENLIT puts the constant value (however many words) on the ITERLST (to become a generated literal) and puts the program location of the literal back on the SCONSLST where the trigger used to be. Note the peculiarity about string constants that the character count is output on the LITERLST (and thus in the generated code) of the string itself, but the official address of the constant is where the string begins. The character count is always in the word preceding the "location" of a string constant.

SCREXLST Script Expression List

A temporary scratch list used only in Pass 1 to collect and process subscript expressions. When an array element is referenced Pass 1 collects the subscript expressions (separated by commas) and calls Combine Subscripts in Pass 0 to produce a single subscript expression or constant. To do this, COMBSCRS uses the information on the SYMTABLE, ARRAYLST, and GROUPLST, as well as that on the SCREXLST. The DIMENSION statement also uses the SCREXLST to collect the lower bounds of dimension ranges for each subscript, and then calls Derive Offset (also in Pass 0) to compute the array offset. See "Subscripts", 3.3.

SCRPTLST Subscript List sometimes called Script List)

Entry Size = 2 Key Size = 1

0	Subscript expression pointer
1	Temp pointer into which expression is to be stored

This list is used throughout Pass 2 to accumulate the subscripts that cannot be computed in-line. For example, if we need to address ARRAY(I+J) in the middle of an expression, we just compute I+J in an index register and use it. But if we need ARRAY(SQRT(X)), it is not convenient to drop the calculation we are already doing and go off and compute a square root.

Therefore, we push the calculation of this subscript to the beginning of the statement. The procedure for doing so is called Retroactive Subscript Generation.

What we do is register the expression pointer on the SCRPTLST and decide on a temp for it to be stored in. Then we go ahead and behave as if this value were already available in the temp. Then at the end of the statement generation, we check to see if there is anything on the SCRPTLST. If so, we delay all the code for the statement onto the DCODLIST, then generate the subscripts required, then put the code back.

This may be a multi-level process. Suppose we have ARRAY(SQRT(ARRAY(SQRT(X)))). Then computing the outer subscript involves an uncomfortable inner subscript. This expression will then show up on the SCRPTLST and must be generaged ahead of <u>all</u> the code in the statement, including the other subscript code. We speak of the subscript as having had "children". See also "Retroactive Subscript Gen (SCRPTGEN)", 2.14.4.

SIGNLIST Sign List

This is a temporary scratch push-down list used in Pass 2 arithmetic expression generation. It keeps track of whether the current piece of an expression is being generated with the right sign or the wrong sign. A TRUE value (all 1's) indicates the wrong sign, while a FALSE (all zeroes) means correct sign.

Quite often signs are updated by Exclusive-ORing them together. In fact, many of the generation routines exit through Update Sign And Exit, which consists of the following.

```
UPSIGNEX  MOF   SIGNLIST

          EOS   *SIGNBOT

          BRL   EXITML
```

And that is the main reason there is an EOS POP. More complicated sign logic occurs in Sum Gen (SUMGEN) in which the bottom two signs may be exchanged. This happens when Pass 2 decides to generate two terms backwards.

SPLBLIST Special Label List

This is a temporary scratch list used only by Pass 2 in generating logical expressions. The routines EXPJUMPT (Logical Expression Gen Jump If True) and EXPJUMPF (Logical Expression Gen Jump If False) need a label

to generate the Jump(s) to. This label is the bottom entry on the Special Label List. It is quite often a generated label, but in the case of a Logical IF statement controlling a GO TO, it may be a statement label. The reason this is a list instead of just a temp is that subexpressions may have their own labels to jump to.

Many of the comments in Pass 2 refer to this list as the "SPEC LABEL LIST".

SRCLNLST Source Line Code List

Entry Size = 2 Key Size = irrelevant; list is not searched

0		Source Line part with Debug Gen Indicator
1		Label Pointer part (statement or created)

This is another one of those delayed code lists associated with the FORTRAN Debug Package (the FDP option) that helps the user debug his program. The other delayed code lists are the ENTRYLST and the LABELLST. Every time a source line number is received from Pass 1, a two-word item is generated on this list. This is in fact perfectly valid code that could go onto the CODELIST but the FDP wants it all together in a table at the end of the program, so it is saved till later (see 3.19.2).

The list is used only in Pass 2. Items are put onto the list by a routine called REGSRCLN and are dealt with at the end of compilation in a routine called DBTBLGEN (Debug Table Gen).

SUMLIST Sum List

A temporary scratch list used only in Expression Scan (in Pass 1) to collect the operands of one or more consecutive "+" (addition) operators.

SUMRYLST Summary List

Entry size = 2 Key size = 0 (i.e. 1, but lower twenty bits only)

0	YADDRESS	Address (displacement) within appropriate area
1	YSYMPNTR	Pointer to symbol (in SYMTABLE)

This is a temporary scratch list used only during the printing of the functional groups in the summary listing in Pass 3. (See "Summary Listing", 2.30.) A class of symbols, such as the local variables, is extracted from the SYMTABLE and put on the SUMRYLST so that it can be sorted into order by address. The key size of 0 specified for this list (which is normally used for labels) assures that the sort will only sort on the address field, ignoring any entraneous bits in other fields.

SVERRLST Save Error List

Entry size = 2 Key size = 1

This list has exactly the same format as the ERRORLST, because it is used to save for later use information that has been collected on the ERRORLST. This happens in two areas. When a statement has to be scanned in more than one way to find out what it is (e.g. DO3K=1) the previous error messages are saved on the SVERRLST while the new scan is taking place and possibly creating new error messages on the ERRORLST. Tests are then made to decide which set of error messages is appropriate. (See "Overall Statement Scan and Fail Structure", Section 2.7.2.) The other place the SVERRLST is used is in Release Card (see Section 2.3.2). When printing out the error messages that have been collected, if more than one marker needs to be printed at the same character position (or several character positions have multiple markers), the extra message pointers are saved on the SVERRLST and processed in succeeding sweeps through the list; each sweep produces a line of markers.

SYMTABLE Symbol Table

Entry size = 6 Key size = 2

0	YSYMBOL (YNAME)	First half of name
1		Second half of name
2	YSYMBID	Symbol ID (class, type, trait flags)
3	YSYMXTRA or YINTRNR	Extra information (pointer to ARRAYLST) -or- Intrinsic function number
4	YALOCPTR or YPERMWRD	Allocation pointer -or- Permissibility word
5	YALODISP or YINTRCL	Allocation displacement -or- Intrinsic function type and argument class

All symbols used in the source program are kept in this list, except for the names of labeled COMMON blocks, which are kept on the BLNAMLST. Some symbols may have more than one entry. For example, the name of a FUNCTION appears both as a subprogram and as a scalar (in which to return the result fo the FUNCTION). Also, each statement function dummy is a separate entry from any other symbols that may have the same name.

The symbols are registered on the SYMTABLE in the order they are encountered, and this order is maintained until the summary listing, when the list is sorted into alphabetic order. After that sort, symbol pointers are no longer valid.

Most symbol pointers do not have the SYMTABLE list number in the upper eight bits. Instead, once they have been classified (with the REG POP), they adopt the symbol's class (e.g. SCALAR, SPROG, etc) from the class field (upper eight bits) of the ID word (YSYMBID). Symbol Table pointers therefore have to be specially recognized. They also have to be specially treated because the traits (tested by the SOF POP) are not found in the Code word, as is true of other lists, but in the lower twenty bits of the ID word item in the symbol's entry.

The Symbol Table is expected to become the largest list (aside from the Output List), so it is allocated as the first ordinary list, with plenty of room to grow. (See "Initialize Lists", 1.4.8.)

The usage of the last three words in each SYMTABLE entry depends on what pass is operating and on what kind of entity the symbol represents (scalar, array, subprogram, etc.). All of these, as well as the class and trait values in the ID word, are explained in Section 3.1, "Identifiers".

SYTCHLST Symbol Table Change List

Entry size = 2 Key size = 1

0		Pointer to changed value (on some list, usually SYMTABLE)
1		Original value

During the scanning of statements, information may be learned about symbols (and labels) that results in changes to the Symbol Table (or a label list). For example, the REG, SUP, and SUL POPs change bits in already established entries on a list. If the scan fails and has to back up and try again, this new information must be discarded. The normal list saving techniques (with SAL) will not work, because they only deal with new information tacked onto the bottom of lists. Therefore, whenever an old value is changed, the original value and its location are saved on the SYTCHLST, so that they can be restored if need be. The REG, SUL, and SUP

POPs do this automatically (sometimes even when you do not want them to, such as when Pass 2 needs to REG the EXTREF trait). The information is restored by FAIL (see "TRY-Fail", Section 2.7).

TERMLIST Term-List

This is a temporary scratch and manipulation list whose heaviest use is in Pass 2. It is the main place that one pulls plexes to. For example, the terms of a sum or product are pulled here to be worked on for code generation. In all cases that I can think of, the items on this list are one word apiece.

TRYLIST TRY List

Entry size = 2 or 4 (defaulted to 1 in Code word)

0*		Old character count*
1*		Old value of FEXEXIT*
2		Delayed Fail Flag (TRUE = delayed fail)
3		Automatic Succeed Flag (TRUE = TRY cannot return FALSE)
* Present only on statement level scans (under FEX), not on TRYs		

This list is used in Pass 1 to save recursive information that is needed during scans that may fail and have to be rescanned. At the statement level, all four words are used. Each TRY uses only a two-word entry. There is no confusion caused by the different size entries, because each level takes care of its own information before returning to any other level. (See "Overall Statement Scan and Fail Structure", Section 2.7.2, and "The TRY POP", Section 2.7.4.)

UNITLIST I/O Unit List Static

Entry Size = 3 Key Size = 1

0	YUNITNO	Unit number (as integer)
1	YUNITRW	Read/write bit (1 for read, 2 for write)
2	YUNITLNK	Unit link address

This is a static list of seven items, corresponding to the seven input/output unit numbers that are given special attention by the compiler. This list is used only in Pass 2, and the routine to consult is CHEKUNIT. This list exists because we have to create external REFs to such DCBs as F:108 in order to get the characteristics that we want for our standard FORTRAN units.

The actual contents of this list are in the assembly called P2P3ROOT. Also, the link constants addressed by YUNITLNK are in P2P3ROOT.

WORKLIST Work List

This is the principal list in the compiler. It serves as an accumulator (or register) for POP code and as a scratch list for manipulating many different things. Most arithmetic, building, and testing take place on the WORKLIST. There are numerous POPs that automatically use the WORKLIST for their "register". To facilitate addressing of the bottom several items on the WORKLIST, they have been given special names and are handled by a whole special POP addressing scheme (described under "POPs").

The bottom item on the WORKLIST is called W0, and the next four are W1, W2, W3, and W4. (See also "A few basic lists", Section 1.4.6.)

XCONSLST Hexadecimal Constant List

Entry size = 2 Key size = 1 Mode = Integer

Traits: Addressable, Constant, Signed Addressable, Test Addressable

0	YCONVAL	Value of constant
1	YIHXUSED or	Constant used trigger (TRUE = used) -or-
	YIHXLOC	Constant literal location (relative to "P" origin)

All hexadecimal constants are kept on this list. They could be kept on the ICONSLST, since they are integer, but this tells the object listing to show them in hexadecimal. Most hexadecimal constants are specified either with nZxxxxx or with X'xxxxx' (on in-line assembly lines). Some, however,

are created by the compiler, when it combines integer mode constants. If the combined constants are all truly integer (i.e. on the ICONSLST), the result is registered on the ICONSLST and thus listed in integer. If any of the constants is not truly integer, the result is registered on the XCONSLST and thus listed in hexadecimal. For example, observe the difference in the following two statements.

```
J = 4 * 3          J = 4 * 1RW

  LI,9  12             LI,9  X'398'

  STW,9 J              STW,9 J
```

Clearly 4*1RW expressed in any other way (e.g. Hollerith) would not be very meaningful.

XREGLIST Index Register List (Static)

Entry Size = 6 Key Size = 1

0	YINDWHAT	Index contents pointer (expression)
1	YINDADND	Addend
2	YINDBUSY	Index register busy trigger
3	YINDEXNR	Index register number (as integer 1, 3, 5, or 7)
4	YINDEXX	Index register number in X-Field
5	YINDEXR	Index register number in R-Field

This is a static list of exactly four items--one for each of the odd-numbered index registers. We call the odd-numbered ones X Registers (as opposed to the even-numbered ones which are DO-Registers) because they can carry out the full range of integer arithmetic operations which even registers cannot.

This list is used only in Pass 2 where it keeps track of the contents and status of these registers. Some of the information in this list is constant (the register numbers) while the rest is variable. See 2.17.1.

The actual contents of this list are in the assembly called FORTTEMP.

ZEROLIST Zero List (Static)

This is a list that is always empty and whose base is location zero. It is also list number zero. It was put in so that addresses with zero in the upper portion could be treated as if they were list pointers (e.g., with POX).

This feature is used by OUTBLC and PROPERND in Pass 3 (see Section 2.29.3.3, Paragraph 5, and Section 2.29.3.3.1.3).

1.6 POPs

Most of the compiler is written in an interpretive language called POPs. There are several reasons for this.

1. It allows us to define a very powerful instruction set that is specifically oriented to the things our compiler needs to do. For example, there are POPs to do such things as: manipulate information on lists (which are dynamic and therefore constantly moving around in memory); scan source input; generate object code; test the characteristics of symbols and constants; create diagnostics; print in alphanumeric, hexadecimal, or decimal. All of these things could, of course, be done by subroutine calls, but each would require two words (one for the call, one for the operand) instead of the one word used by a POP.

2. The POP language can be quite machine-independent, and thus transferable from one machine to another. Most of Pass 1 (which scans source input and converts it to internal notation, and is thus not very concerned with what kind of object code will have to be generated) was lifted verbatim from the 9300 FORTRAN IV compiler and just reassembled with a new POP Proc.

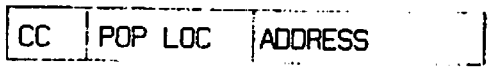
3. The readability of the compiler listings is improved. Once one is familiar with the POP instruction set, it is just like working on a different (and rather interesting) computer. The POP mnemonics quickly come to be more meaningful than longer sequences of code, or subroutine calls, that do the same thing.

4. The code becomes recursive (can call itself) almost automatically, since most information is processed on push-down lists and calls are made with the JRS (Jump Recursively to Subroutine) POP.

5. The POP structure allows us to add a new feature to the basic instruction format, namely the true and false conditions. Many POPs and subroutines return answers, analogous to the Sigma condition codes. Whereas Sigma provides only a branch on certain conditions, the compiler can conditionally execute any POP based on the answer flag. It is as if there were a STWLZ (store word if less than zero) instruction (and many others like it). This often eliminates having to branch around short segments of conditional code.

POPs stands for Programmed Operators. These were a standard hardware feature on the XDS 900-Series computers. They had the same format as regular instructions, but a special bit indicated that the op-code was not to be treated as an op-code, but rather as a location in the POP jump table, which contained the addresses of the subroutines that were to process the POPs. For example, the POP numbered 23, when executed, automatically took you to location 23 in the POP jump table, which took you to a subroutine that did what that POP was supposed to do. Since this was a hardware feature that took place automatically whenever the POP bit was set, it was perfectly acceptable to intermix POP code and machine code. The POP simply acted as a one-word subroutine call with argument.

POPs are not a hardware feature on Sigma, but the INT (Interpret) instruction was specifically designed to simulate that feature. It operates on "instructions" that have the following format:

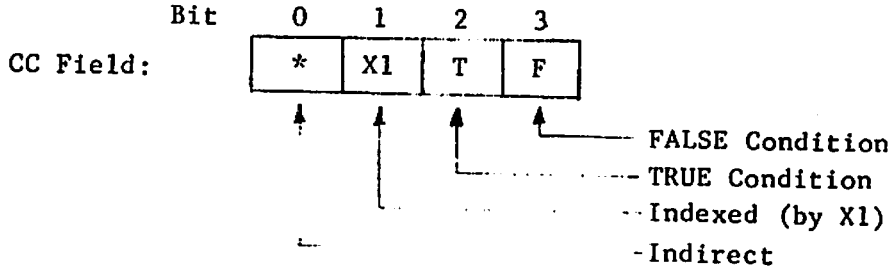


4 Bits 12 Bits 16 Bits

Since the POP location field is 12 bits, a jump table is not required, as long as the POP subroutines are within 4K words of each other. The interpreter simply branches directly to POPLOC relative to a base location (POPBASE in the assembly INTERPTR).

Note that the address field is only 16 bits, instead of the usual 17. This means that POPs can only directly address the lower 64K of memory (see "64K Restriction", Section 4.5.6). One corollary of this is that indexed POPs cannot have a negative address (e.g. FET -1,X1), because there is no way to distinguish a small negative value (which needs to be sign extended to 17 bits) from a large positive value (which must not be sign extended). Another corollary is that a POP cannot address indirectly through another POP, since the last bit of the POP field will be included in the 17- bit address used to determine the indirect address. If we made either indexing or indirect addressing work only on 16 bits (which we could do manually in the interpreter), then there would be no way to address data above 64K. As it is, all the lists can be anywhere in memory, becasuse they are always addressed either indirectly or with indexing.

The CC field is place in the Condition Codes by the INT instruction, and we are free to do whatever we like with it. What we do is the following:



The indirect bit is the same as in regular Sigma instructions, but all of these special conditions are handled manually by the interpreter. A POP may have any combination of conditions, except that both T and F may not be specified. If both indirect and indexing are specified, the interpreter does the indexing before the indirect, which is exactly the opposite of the Sigma hardware, which does indexing after indirect. This is mainly a holdover from (the 9300 hardware did indexing first), rather than a feature that the compiler makes much use of.

The following example of POP code illustrates some of these features:

```
LXI     N          Load N into index 1

FET     *TABLE,X1  Fetch to WORKLIST the word addressed by Nth location
                                                          in TABLE

SWG     VALUE      Test if greater than VALUE

ADW,T   =1         If so, add 1

GET,F   VALUE      If not, replace word with VALUE

MET,F   VFLAG                 and set VFLAG to .TRUE.

MON     TERMLIST   Move result onto the bottom of the TERMLIST

BRU     EXIT       Exit from this subroutine (and this recursive level)
```

There are a great many POPs (about 150). They are all tabulated, both alphabetically and by function, in Section 1.7. Some are relatively analagous to hardware instructions (manipulating and combining values, testing, and branching) while others are far more esoteric and compiler-oriented.

1.6.1 True/False Feature

Every POP can be written three ways: 1. execute unconditionally; 2. execute if the Answer Flag is True; or 3. execute if the Answer Flag is False. Some POPs exist only to return an answer--i.e., to ask a question. In addition to certain POPs returning answers, certain subroutines return answers, too.

The convention is, when calling a subroutine that returns an answer, to use the POP code QJS (Query Jump to Subroutine) rather than the ordinary JRS (Jump Recursively to Subroutine). In fact, these two POPs are the same POP and do the same thing. The QJS mnemonic merely informs the reader of the listing that the subroutine called is expected to return an answer. This convention is not followed with a great deal of consistency in Pass 1, much of which remains unchanged from the 9300 compiler. Passes 2 and 3 use it fairly consistently.

Code is usually written as if there were only one answer flag. But in fact there are many: one for each level of recursion (i.e., one for each word on the EXITLIST). This costs very little and adds quite a convenience to coding in POPs. It is possible to make a test, then call a lower level routine that does some testing of its own, then return knowing that the result of your original test is still intact.

In the 900 and 9300 Series compilers, there were only two POPs that could test the answer flag. These were JAT (Jump if Answer True) and JAF (Jump if Answer False). In the current compiler the True or False option can be applied to any POP, and this is done by appending ",T" or ",F" to any POP mnemonic. The T and F become the second part of the Command Field (i.e., CF(2) in Meta-Symbol terminology). If the T or F is omitted, the POP is executed unconditionally.

Some nice things can be done by attaching T and F tags to POPs that themselves return answers. For example, consider the sequence

```
TMT       TRIG1

TMT,F     TRIG2

TMT,F     TRIG3
```

This will set the Answer Flag True if any of the three triggers TRIG1, TRIG2, or TRIG3 is true. Similarly, a kind of ANDing can be achieved:

```
TMT       TRIG1

TMT,T     TRIG2

TMT,T     TRIG3
```

delivers an answer TRUE only if all three triggers are true.

The kind of coding just described is fairly widely used in the compiler, and the only real danger with it is that, if it is not well commented, it can be obscure and hard to understand. Some of this code is well-commented; some is not. Most of it is quite efficient.

1.6.2 Special Addressing

In addition to indexing and indirect addressing in POPs*, there is a feature for addressing the bottom few items on the WORKLIST and the PARAMLST. If one wants to address W0, he can write "*WORKBOT". In fact, if the special address W0 is used, the POP generating Proc (in SYSTEM FORTCOMP) automatically generates an indirect bit and a reference to WORKBOT.

It is often desirable, however, to go farther in addressing the WORKLIST--to address W1, W2 and so forth. This cannot be done automatically using indexing and/or indirect addressing because there is no cell containing one or two less than what WORKBOT contains. So we have a special feature for handling these addresses.

When the POP Proc sees, say

```
FET    W1
```

it does not put the FET op code into the op code field. Instead it puts the WAD op code (WORKLIST Address Direct) into the op code field. It then divides the operand field up into two fields--one containing the number 1, indicating that it is W1 that is wanted, and the other field containing the FET op code. So the POP structure is effectively turned upside down, for the op code becomes the operand and the operand becomes the op code. The same technique is used for W2, W3 and W4.

The symbols W0, W1, W2, W3, and W4 are defined as two-item lists (in the Meta-Symbol sense) so that the POP Proc can distinguish them from normal operands.

---

* Remember: when indexing and indirect are both specified, the convention is the rev s hardware. On POPs, indexing is done before indirect. On the hardware, the indire t

The same feature is provided for addresses on the PARAMLST. Here it is provided in two flavors--direct and indirect. Often the items on the PARAMLST are addresses put there by the ADR POP, so what is most often desired is *P0, *P1, *P2 and *P3. These are provided, as are P0, P1, P2, and P3. An alternate way of writing the indirect versions is without an asterisk but with the letters "IND" appended to the symbol, as in P0IND, P1IND, etc. This is a carry-over from the 9300, whose version of Meta-Symbol would not allow us to detect an asterisk on a list.

The indirect addresses are provided by the PAI POP (PARAMLST Address Indirect). The direct addresses are provided by the PAD POP (PARAMLST Address Direct); P0 is not special-cased to generate *PARAMBOT; the case was thought to be too rare to bother with. PAD works just fine with a zero value. WAD would work just fine that way too, if it were ever generated.

The three POPs just discussed (WAD, PAD, and PAI) are all described in the section on Addressing POPs, Section 1.7.2.10. This category also includes ADR, EAT, EAX, DXI, FEX, and XEQ.

1.6.3 The POP Procs

There lives in SYSTEM FORTCOMP a set of three Procs. These handle the generation of the POP instructions. The first Proc has 154 different names, the second has one name, and the third has two.

The big Proc handles the bulk of the work. It checks for illegal constructs in the operands and can print the diagnostic "POP SYNTAX". It handles the True/False feature (Section 1.6.1) and special addressing (Section 1.6.2). It detects the use of special addresses such as W0, P1, etc. by the fact that these symbols are defined as two-element lists. These definitions are also in SYSTEM FORTCOMP. A few are shown here as examples.

```
W1        EQU  (3,1)

P2        EQU  (1,2)

P3IND     EQU  (2,3)

W0        EQU  (4,0)
```

The Proc makes a special case out of W0, generating an indirect reference to WORKBOT. For all the other special addresses, the Proc turns the POP around, putting WAO or PAD or PAI in the op code portion and putting the POP's op code and the displacement in the operand portion. Of the sixteen bits of the operand portion, the first four are the displacement from the list bottom (expressed as a positive number or zero) and the last twelve bits are the op code.

The QIX POP has a Proc all its own. This is because its operand is not an ordinary address but a pair of eight-bit numbers. These numbers represent a range. (See the description of the QIX POP, Section 1.7.2.9.) Since special addresses are already represented by two-element lists, we represent ranges by three-element lists, so that they can be distinguished. The third element is zero and is ignored, except to check that it is there. The Proc allows indexing and special addressing, but it disallows indirect addressing. It also complains if the argument is not a list.

The third Proc is for JAT and JAF and is the simplest of all. It does not allow special addressing, nor does it permit there to be a CF(2). CF(2) is the second part of the command field, where you would write T or F. If any of the restrictions are violated, it gives the warning "POP SYNTAX". All the error messages given by the three POP Procs have a severity of zero.

1.6.4 Naming Conventions

To say that we have any conventions at all about naming our POPs seems almost ludicrous. But there are some conventions, some of them dating back to the SDS 920 FORTRAN II compiler. The conventions all have to do with the initial letter, and every rule has at least one exception.

B. B almost always stands for "Build" and means the code-generating POPs. All the code-generators start with B except CNL (which is a sort of fringy code generator). There are only two POP names beginning with B that are not code generators: BAM and BRU. BAM is often used to address a table of code-generation POPs.

C. C can stand for "Copy", "Character", "Count", or "Comment".

F. F can stand for "Fetch", "Fail", "File", or "Flip".

G. G stands for "Get" except for GNL where it means "Guarantee".

J. J stands for "Jump". Always. This is the exception to the rule that every rule has at least one exception.

M. M usually means "Move" but can also mean "Memory" or "Multiply".

P. P often means "Print" but can also mean "Pull", "Process", "Pointer", "Plex", or "Parameter".

Q. Q means "Query" except in the QSA POP where it stands for "Quote". All the Q POPs return an answer. We picked up this Query idea from Digitek who, in one compiler, had all their answer-returning POPs begin with Q. This was fairly late development on our part, however, and it is by no means true that all our answering POPs start with Q. Many start with S, for "Set answer flag", and a few start with T for "Test".

S. S is by far the most heavily used initial letter. Thirty-five POP mnemonics begin with the letter S. Nineteen use it to mean "Set", of which seventeen use the word in the sense of setting the answer flag, and two use it in the sense of "set up". Three mnemonics use it to mean "Subtract", while two use it to mean "Search". Three use it for "Store" and two for "Shift". Then we have a couple meaning "Save", two meaning "Selective", and a couple of oddballs meaning "Sort" and "Skip".

To develop a feeling for these naming conventions, the reader should read the above with Table 1.7, the alphabetical table of POP mnemonics.

1.7 Tables of POPs

Following are two tables of the POPs in the compiler. The first is alphabetic and gives only the POP mnemonic, its full name, and its functional category. The second contains descriptions of the individual POPs, bu functional category. The alphabetic table is useful for looking up a particular POP that you are not familiar with. The functional table is preferable for learning about the POP language, what it can do, and how. Each functional category also lists the names of related POPs in other categories. For example, CSA (Character Scan with Answer) is certainly a "test" POP, but more properly belongs in the "Input Scan" category.

For each POP, the following information is given:

1. The POP mnemonic. A few POPs also have an alternate mnemonic, which is shown.

2. The full name of the POP.

3. Offset to the left, under the POP mnemonic, is a number that indicates the POP's effect on the size of the WORKLIST. Almost half the POPs automatically deal with the WORKLIST, so it is useful to know whether they increade its size (e.g. +1), decrease it (e.g. -1), or leave it alone (0). If they do not affect its size, but they can change the value in W0 (the bottom word on the WORKLIST), an asterisk precedes the zero (*0). Sometimes the effect on the WORKLIST depends on the answer returned by the POP (e.g. +1 T, 0 F).

4. What can be affected by the POP. If the POP returns an answer, then this is always listed as the first thing that can be affected, with an indication of what condition causes the answer to be TRUE. Other things that may be affected include: lists; memory locations; W0; the X1 and A registers; the control location (i.e. on jumps); the print buffer position; the SI and LO files, etc.

5. A description of the POP and its usage. In these descriptions (and in the above list of things affected), the symbol "e" is used to represent the effective address of the POP. Thus, for example, the EMP POP empties list e; FET fetches the contents of memory location e.

1.7.1 Explanation and Alphabetic Table

The POPs have been arbitrarily divided into sixteen categories for ease of explanation. The following table is an alphabetical list of the POPs, telling what the mnemonic means, what category it belongs to, and the number of the category. If you want to know what the POP <u>really</u> does, turn to the appropriate subheading in 1.7.2 and you will find it. The POPs in the categories are not listed in alphabetical order: they are listed in an order that is easy to read and understand, because certain write-ups depend on previous write-ups for understanding. The sixteen POP categories are as follows.

P O P C A T E G O R I E S

1. MANIPULATION
2. ARITHMETIC
3. BOOLEAN
4. MEMORY MODIFICATION
5. JUMP
6. TEST
7. GENERAL LIST
8. SPECIFIC LIST
9. POINTER
10. ADDRESSING
11. PLEX
12. PRINT
13. INPUT SCAN
14. GENERATING
15. CONTROL
16. DEBUG

Table 1.7.1 ALPHABETIC TABLE OF POPS

Mnemonic and name	Category	Category Number *
ADR Address to Parameter list	Addressing	10
ADS AD to Storage and discard	Arithmetic	2
ADW AD to WO	Arithmetic	2
BAF Build Absolute instruction and File	Generating	14
BAFR BAF with R-temp register field	Generating	14
BAM Build by Address Mode (actually Execute by Address Mode)	Control	15
BIF Build Instruction with Full addressing capability	Generating	14
BIFR BIF with R-temp register field	Generating	14
BIM Build IMmediate instruction	Generating	14
BIMR BIM with R-temp register field	Generating	14
BIN Build Instruction with Normal address	Generating	14
BINR BIN with R-temp register field	Generating	14
BLC Build Library Call (BAL,6)	Generating	14
BNG BriNG (same as GRP--Get Relative to Pointer)	Pointer	9
BOP BOttom Pointer	Pointer	9
BRL BRanch and Leave interpretive mode	Jump	5
BRO Build with Register Operand	Generating	14
BROR BRO with R-temp register field	Generating	14
BRORR BRO with R-temp register field and address		
BRU BRanch Unconditional	Jump	5
CAE Copy and Empty	General List	7
CAR Copy and Release	General List	7
CIC Copy Inherit and Count	Plex	11

CIF	Copy Inheriting First-term traits	Plex	11
CNL	Comment Next Line	Generating	14

* Category Number is the value n in the section number 1.7.2.n

CNT CouNT a list	General List	7
COG COpy Group	Specific List	8
CPY CoPY	General List	7
CSA Character Scan with Answer	Input Scan	13
CSF Character Scan or Fail	Input Scan	13
CSK Character Scan with answer and Keep	Input Scan	13
DTN Debug Trap eNter	Debug	16
DTX Debug Trap eXit	Debug	16
DVW DiVide WO by memory	Arithmetic	2
DXI Decrement indeX Immediate	Addressing	10
EAT Effective Address to Temp (EATEMP)	Addressing	10
EAX Effective Address to indeX	Addressing	10
EMP EMPty list	General List	7
EOS .EOR. (Exclusive OR) to Storage and discard	Boolean	3
ERC ERror on Current character	Specific List	8
ERL ERror on Last active character	Specific List	8
ERW ERror on character specified in WO	Specific List	8
ETW ExTract to WO (.AND. WO)	Boolean	3
FAD File ADdress	Specific List	8
FET FETch to WORKLIST	Manipulation	1
FEX Fail EXit	Addressing	10
FIC Fixed-length plex Inherit first-term traits	Plex	11
FIL FIle on CODELIST	Specific List	8
FIP Fixed-length plex Inherit from all traits	Plex	11

FLIP FLIP list upside-down	General List	7
FRP Fetch Relative to Pointer	Pointer	9
FTE Fetch To EATen list	Manipulation	1
GET Replace WO	Manipulation	1
GNL Guarantee N lines on LO page	Print	12
GRP Get Relative to Pointer	Pointer	9
JAF Jump if Answer FALSE	Jump	5
JAT Jump if Answer TRUE	Jump	5
JDX Jump and Decrement Index	Jump	5
JPN Jump and Process Next item	Jump	5
JRS Jump Recursively to Subroutine	Jump	5
LDA Load A register	Manipulation	1
LOD LOaD a list	General List	7
LODI LOaD list from Input	General List	7
LX1 Load Index 1	Manipulation	1
MCO Move Central (1 and 2) Onto list	Manipulation	1
MEF MEmory FALSE	Memory Modification	4
MET MEmory TRUE	Memory Modification	4
MMO Memory Minus One	Memory Modification	4
MOF Move OFf bottom and onto WORKLIST	Manipulation	1
MOK Move on and Keep	Manipulation	1
MON Move Onto from WORKLIST	Manipulation	1
MOR Move Off and Release if empty	Manipulation	1
MPO Memory Plus One	Memory Modification	4
MPS MultiPly WO and Store (Multiply to Storage)	Arithmetic	2

MPT Memory Plus Two	Memory Modification	4
MPW MultiPly WO	Arithmetic	2
NUL NULl	Control	15
ONE Store ONE (One to Memory)	Memory Modification	4
ORS OR to Storage and discard	Boolean	3
PAC Print Alphanumeric with Count	Print	12
PAD Parameter Address Direct	Addressing	10
PAI Parameter Address Indirect	Addressing	10
PAM Print Alphanumeric, Minimum width	Print	12
PBL Print BLanks	Print	12
PDC Print Decimal with Count	Print	12
PDM Print Decimal, Minimum width	Print	12
PHC Print Hexadecimal with Count	Print	12
PLO PLex Open	Plex	11
PNI Process Next Item	Pointer	9
POX POinter to indeX	Pointer	9
PRC PRint Character	Print	12
PRL PRint Line(s)	Print	12
PRQ Print Qote	Print	12
PUG PUll Group	Specific List	8
PUL PUL a plex	Plex	11
QBL Query Branch and Leave interpretive mode	Jump	5
QIX Query ID field to indeX	Pointer	9
QJS Query Jump to Subroutine	Jump	5
QMN Query Memory Negative	Test	6
QSA Quote Scan with Answer	Input Scan	13

QZR Query memory ZeRo	Test	6
RAD Reserve and ADdress	General List	7
REG REGister	Specific List	8
RGI ReGister Integer constant	Specific List	8
RLS ReLeaSe	General List	7
RPL Reserve and PuLl	Plex	11
RST ReSTore a word	Specific List	8
RSV RsSerVe a list	General List	7
SAL Save A List	Specific List	8
SAV SAVe a word	Specific List	8
SAW Shift Arithmetically WO	Arithmetic	2
SED Set if WO Equal and Discard	Test	6
SEK Set if WO Equal and Keep	Test	6
SER SEaRch a list	General List	7
SERK SEaRch with Key	General List	7
SGD Set if WO Greater and Discard	Test	6
SKR SKip Reduce (reduce and skip if negative)	Memory Modification	4
SLW Shift Logically WO	Arithmetic	2
SME Set on Masked Equality	Test	6
SNE set if list Not Empty	Test	6
SOC Set On Character	Input Scan	13
SOF Set On Flag (trait)	Test	6
SOL Set On Label	Specific List	8
SOM Set On Mode equality with WO	Test	6
SOP Set On Permissibility	Specific List	8

SOR Set On Range	Test	6	
SORT SORT a list	General List	7	
SOT Set on Test (Set on WO ID field equality)	Test	6	
SRC Set on Range of Central	Test	6	
SRV Set if list ReserVed	Test	6	
SSK Selective Store and Keep WO	Boolean	3	
SST Selective STore and discard WO	Boolean	3	
STA STore A register	Manipulation	1	
STK STore WO and Keep	Manipulation	1	
STO STOre WO and discard	Manipulation	1	
SUFS SUbtract From Storage and discard WO	Arithmetic	2	
SUL Set Up Label	Specific List	8	
SUP Set Up Permissibility	Specific List	8	
SUS SUbtract from WO and Store and discard	Arithmetic	2	
SUW SUbtract from WO	Arithemetic	2	
SWA Set if WO .AND. memory is non-zero	Test	6	
SWE Set if WO Equal to memory, discard if equal	Test	6	
SWG Set if WO Greater than memory	Test	6	
TAB TABulate print buffer	Print	12	
TMT Test Memory TRUE	Test	6	
TOR Take Off top and Release if empty	Manipulation	1	
TOT Take Off Top	Manipulation	1	
TPT Top PoinTer	Pointer	9	
TRY Try	Jump	5	

TZR	Test ZeRo (alternate mnemonic to QZR)	Test	6
UNR	UN-Reserve a list	General List	7
WAD	WORKLIST Address Direct	Addressing	10
XCH	eXCHange with WO	Manipulation	1
XEQ	Execute	Control	15
ZAP	ZAP (wipe out a list completely)	General List	7
ZRO	store ZeRO	Memory Modification	4

1.7.2 Functional Table of POPs With Descriptions

1.7.2.1 MANIPULATION

FET FEtch to WORKLIST

+1 Affected: WORKLIST

The contents of memory location e are appended to the bottom of the WORKLIST. Since most operations are done in W0, this is the principal "load" POP in the compiler. However, for a simple "load-store" (no other manipulation), LDA is preferable. See also GET and FTE.

GET Replace W0

*0 Affected: W0

Exactly like FET, except the fetched word replaces W0 instead of being appended to the WORKLIST. Thus GET is much like the typical hardware load instruction, which destroys the previous contents. This POP should not be executed if the bottom file of the WORKLIST is empty, since there is no valid W0 to be replaced.

STO STOre W0 and discard

-1 Affected: WORKLIST; Memory location e

The contents of W0 replace the contents of e, and W0 is then removed from the WORKLIST. This is the converse to the FET POP. It should not be executed when there is no W0 (bottom file of the WORKLIST is empty).

STK STore W0 and Keep

0 Affected: Memory location e

Same as STO, but W0 is not discarded. If GET is like the Sigma LW, then STK is like Sigma's STW. STK should not be executed when W0 does not exist.

XCH EXCHange with W0

*0 Affected: W0; Memory location e

Exchange the contents of e with the contents of W0. Often used to exchange with other items on the WORKLIST; e.g. XCH W2, which exchanges the first and third words on the WORKLIST. (This special addressing is

done by the WAD POP.) Should not be executed when the bottom file of the WORKLIST is empty.

MOF Move OFf bottom and onto WORKLIST

+1 T Affected: Answer (T if list e not empty); List e; WORKLIST

0 F

MOF takes one word off the bottom of list e and appends it to the WORKLIST. Thus list e is shortened by one word, while the WORKLIST is lengthened by one word. If the bottom file of list e is empty, MOF returns with answer FALSE and neither list is changed. Otherwise it returns with answer TRUE. This is the typical way to process, one at a time, an unknown number of items on some list. When the MOF gives a FALSE answer, there are no more items.

MOR Move Off and Release if empty

+1 T Affected: Answer (T if list e not empty); List e; WORKLIST

0 F

If list e is not empty, MOR works exactly like MOF and returns with answer TRUE. However, if list e is empty, MOR not only returns answer FALSE, but also releases list e (i.e. removes one reserve marker, if there is one -- see RLS POP). In this case, the WORKLIST is unchanged. Thus it is possible to build up a file of items on some list, and then process them from bottom to top using MOR, which will indicate when there are no more and remove the reserve marker.

MON Move ONto from WORKLIST

-1 Affected: WORKLIST; List e

This is the converse to MOF. MON takes one word off the bottom of the WORKLIST and appends it to the bottom of list e. No answer is returned; MON should not be used when there is no WO.

MOK Move On and Keep

0 Affected: List e

Same as MON, but WO is not discarded. Thus MOK:MON = STK:STO.

MCO Move Central 1 and 2 Onto list

0 Affected: List e

The contents of CENTRAL1 and CENTRAL2 are appended to list e. The CENTRAL area is an often used temporary area, and it is convenient to be able to move a two-word item there onto a list (e.g. a symbol onto the SYMTABLE) without having to do two FETs and two MONs.

TOT Take Off Top

+1 T Affected: Answer (T if list e not empty); List e; WORKLIST

0 F

Takes the top word (from the bottom file) of list e and appends it to the bottom of the WORKLIST. If list e is empty, the lists remain unchanged and TOT returns with answer FALSE. Otherwise TOT returns with answer TRUE. There are some special cases of this POP. If list e has not been reserved (i.e. there is only one file), TOT automatically adjusts the Base and Start parameters to coincide with the Top. That is, as the top items are removed, the list shortens from that end, a very untypical thing to happen. This means that pointers to list e will no longer be valid, since the Base has moved relative to the other words. If list e has been reserved, there is also a special case. As TOTs are done, a gap is introduced between the reserve marker and the new "top" word on the list. After the last word is removed from the top (the file is now empty), if another TOT is given, it not only answers FALSE, but it adjusts the Top and Bottom parameters back to coincide with the Start, thus removing the gap. For further information, see "Dynamic Lists", Section 1.4.1.

TOR Take Off top and Release if empty

+1 T Affected: Answer (T if list e not empty); List e; WORKLIST.

0 F

TOR works exactly like TOT if there is anything on the bottom file of list e (i.e. it is not empty). However, if list e is empty, then TOR not only returns with answer FALSE, but also releases list e. (See the RLS POP.) TOR works in the same way relative to TOT that MOR does to MOF.

FTE FeTch to EATen list

0 Affected: List specified in EATEMP

The contents of e are appended to the bottom of the list whose number is contained in EATEMP (the Effective Address Temp). Typically one or more FTEs are preceded by an EAT POP. For example,

EAT SOMELIST is the same as FET GRUNCH

```
FTE     GRUNCH          MON  SOMELIST
FTE     GLOP            FET  GLOP
                        MON  SOMELIST
```

LDA LoaD A

0 Affected: "A" register

Loads the contents of e into the A register. The A register can be used for simple moves (LDA, STA), and is also used to contain the mask in certain Boolean operations such as SST, SSK, and SME.

STA STore A

0 Affected: Memory location e

Stores the A register into location e. This is used only in simple moves (following a LDA), since no other operations produce results in the A register.

LX1 Load indeX 1

0 Affected: "X1" register

Loads the contents of e into X1. X1 is the only index register available to POP code. It can be used either for indexed addressing of POPs or for counting in loops (see the JDX POP). There is no POP for storing X1. See also the DXI, POX, and QIX POPs.

MANIPULATION related POPs:	Category Category:	Category Number:
SST Selective STore and discard WO	Boolean	3
SSK Selective Store and Keep WO	Boolean	3
FRP Fetch Relative to Pointer	Pointer	9
GRP Get Relative to Pointer	Pointer	9
ADR ADdRess to PARAMLST	Addressing	10

1.7.2.2 ARITHMETIC

ADW ADd to W0

*0 Affected: W0; OVFLTRIG Effect: (W0)+(e) W0

Adds the contents of memory location e to W0 leaving the result in W0. ADW may set the Overflow Trigger (OVFLTRIG) if overflow occurs in the signed integer sense. Many POPs set the OVFLTRIG TRUE, but none of them set it FALSE. It must be initialized by MEF. Thus overflow is accumulative.

ADS ADd to Storage from W0 and discard

-1 Affected: Memory location e; Effect: (W0)+(e) e, OVFLTRIG; WORKLIST discard W0

Adds the contents of W0 to memory loaction e. Discards W0. May set the Overflow Trigger OVFLTRIG (see ADW).

SUW SUbtract from W0

*0 Affected: W0; OVFLTRIG Eff ct: (W0)-(e) W0

The contents of the addressed location e are subtracted from the contents of W0, and the difference is placed in W0. SUW may set the Overflow Trigger OVFLTRIG (see ADW).

SUFS SUbtract From Storage and discard

-1 Affected: Memory location e; Effect: (e)-(W0) e, WORKLISLT: OVFLTRIG discard W0

Subtracts the contents of W0 from the contents of the addressed location e, stores the result in e, and discards W0. SUFS sets the Overflow Trigger OVFLTRIG if overflow occurs.

SUS SUbtract from W0 and Store and discard

-1 Affected: Memory location e; Effect: (w)}-(e) e, WORKLIST;OVLFTRIG discard W0

Less useful than SUFS, this POP is the same excpet that its result has the opposite sign from that produced by SUFS. SUS sets the Overflow Trigger OVFLTRIG if overflow occurs.

MPW MultiPly WO as an integer

0 Affected: WO; OVFLTRIG Effect: (WO)(e) WO

The product, formed by multiplying (at integer scaling) the contents of WO by the contents of the addressed location e, replaces the contents of WO. MPW may set the Overflow Trigger OVFLTRIG if overflow occurs in the signed-integer sense.

MPS MultiPly to Storage and discard

-1 Affected: Memory location e; discard WO Effect: (WO)*(e) e, OVFLTRIG; WORKLIST)

Just like MPW except that the product is stored in e instead of WO, and WO is discarded.

DVW DiVide into WO as integer

*0 Affected: WO, A Register; remainder A Effect: (WO)/(e) WO, OVFLTRIG

The contents of WO are divided by the contents of the addressed location e, and the quotient is placed in WO. The remainder is placed in the A Register. DVW may set the Overflow Trigger OVFLTRIG.

SAW Shift Arithmetically WO

*0 Affected: WO

The contents of WO are shifted e places in the arithemetic sense, which means that on a right shift the sign is extended. If the shift count e is positive, it is a left shift; if negative (in the lower eight bits) it is a right shift. SAW does not affect the Overflow Trigger.

SAL Shift Logically WO

*0 Affected: WO

Just like SAW except that zeroes are inserted on a right shift instead of sign bits.

1.7.2.3 BOOLEAN

ETW ExTract to W0 .AND. to W0

*0 Affected: W0

Takes the contents of memory location e and ANDs it with W0, then leaves the result in W0.

ORS .OR. to Storage and discard

-1 Affected: Memory location e; WORKLIST

Takes the contents of memory location e and ORs it with W0, then stores the result in memory location e. It also removes the bottom entry from the WORKLIST.

EOS .EOR. Exclusive OR to Storage and discard

-1 Affected: Memory location e; WORKLIST

Exclusive ORs the contents of W0 with the contents of memory location e and leaves the result in memory location e. It also removes the bottom entry from the WORKLIST. This POP is used almost exclusively in Pass 2 to do sign logic.

SST Selective STore and discard

-1 Affected: Memory location e; WORKLIST

Uses the mask in the A register to selectively store the contents of W0 in the addressed location e. Bit positions of W0 corresponding to 1's in the A register are stored into (e); bit positions of (e) corresponding to zeroes in the A register are left unchanged. W0 is discarded.

SSK Selective Store W0 and Keep

-1 Affected: Memory location e

Just the same as the SST POP above except that W0 is not discarded.

tu

Category:Number:

Category BOOLEAN related POPs:

SLW Shift Logically WO	Arithmetic	2
SME Set on Masked Equality	Test	6
SWA Set if WO .AND. memory nonzero	Test	6

1.7.2.4 MEMORY MODIFICATION

MEF MEmory FALSE

0 Affected: Memory location e

Stores the logical value .FALSE. (i.e, all zeros) in location e. This is used for setting triggers (flags) for later testing by TMT. See also ZRO.

MET MEmory TRUE

0 Affected: Memory location e

Stores the logical value .TRUE. (i.e., all ones) in location e. This is used for setting triggers for later testing by TMT.

MPO Memory Plus One

0 Affected: Memory location e

Increments location e by 1. Does not test for overflow or affect the OVFLTRIG.

MPT Memory Plus Two

0 Affected: Memory location e

Increments location e by 2. Does not test for overflow or affect the OVFLTRIG.

MMO Memory Minus One

0 Affected: Memory location e

Decrements location e by 1. Does not test for overflow or affect the OVFLTRIG.

SKR Reduce and Skip if negative

0 Affected: Memory location e; Control location

Decrements location e by 1, and if the result is negative skips the following POP. This can be used instead of JDX to control loops in which X1 may be clobbered. Occasionally the compiler uses SKR where it should use MMO, knowing that the results will not be negative. This is only a holdover from the 9300 compiler, which did not have an MMO POP.

ZRO Store ZeRO

O Affected: Memory location e

Stores zero in location e. As implemented, this is the same POP as MEF, but they should not be used interchangeably (though occasionally they are). The POP code should not take advantage of the knowledge that .FALSE. is the same as zero.

ONE Store ONE

O Affected: Memory location e

Stores 1 in location e.

MEMORY MODIFICATION related POPs:	Category	Category Number
STO STore WO and discard	Manipulation	1
STK STore WO and Keep	Manipulation	1
XCH EXCHange with WO	Manipulation	1
STA STore A	Manipulation	1
ADS ADd to Storage and discard	Arithmetic	2
SUFS SUbtract From Storage	Arithmetic	2
SUS SUbtract from WO and Store and discard	Arithmetic	2
MPS MultiPly to Storage and discard	Arithmetic	2
ORS OR to Storage and discard	Boolean	3
EOS Exclusive OR to Storage and discard	Boolean	3
SST Selective Store and Keep WO	Boolean	3
SSK Selective STore and discard WO	Boolean	3
QMN Query Memory Negative	Test	6
TMT Test Memory TRUE	Test	6
QZR Query ZeRo	Test	6

1.7.2.5 JUMP

BRU Branch Unconditional

0 Affected: Control location

Branches to location e.

JAT Jump if Answer TRUE an alternate mnemonic for BRU,T

0 Affected: Control location

Jump to location e if current answer is TRUE. JAT and JAF are holdovers from the 9300 compiler, which did not have T and F conditions available for all POPs. The answer flag is recursive and is found in bit 0 of the bottom word on the EXITLIST (1 = TRUE, 0 = FLASE).

JAF Jump if Answer FALSE an alternate mnemonic for BRU,F

0 Affected: Control location

Jump to location e if current answer is FALSE.

BRL BRanch and Leave interpretive mode alternate mnemonic: QBL

0 Affected: Control location; execution mode

Branch to location e and start executing machine language instead of POPs. This is used to call machine language subroutines. The return address remains in register IL, so the subroutine can return to the POP following the BRL by using a BIR,IL INTERP. Subroutines that return answers should be called using the mnemonic QBL, as a documentation aid, but this is not done consistently (or even often) throughout the compiler, because it was a late addition.

JRS Jump Recursively to Subroutine alternate mnemonic: QJS

0 Affected: Control location; EXITLIST

Appends the return address to the EXITLIST and jumps to location e. This is used to call all POP subroutines, which are then executed at the next recursive level. Each level has its own answer flag (kept on the EXITLIST). Return from a POP subroutine is via a BRU to EXIT or to one of the many other POP exit locations (e.g. Clear 1 and Exit TRUE). The mnemonic QJS should be (but is not always) used to indicate that the subroutine will return an answer.

TRY TRY

0 Affected: Control location; EXITLIST: SAVELIST: TRYLIST;

ERRORLST; SYTCHLST

TRY is a tentative JRS used for multiple scanning of source input. It is described in detail in the "The TRY POP", Section 2.7.4. Basically, it behaves like JRS but converts EXIT to EXIT TRUE, and FAIL to EXIT FALSE. In other words, if the JRS succeeds, TRY returns TRUE; if the JRS fails, TRY returns FALSE. It is also possible, however, for a TRY to fail, and to not return at all. The TRY POP itself is mainly coded in POPs.

JDX Jump and Decrement indeX

0 Affected: X1 register; Control location

Decreases X1 by 1. Then jumps to location e if X1 is still greater than zero. If X1 is less than or equal to zero, control falls through to the next POP. For example, a loop can be performed five times with the following control:

```
        EAX  5
2A7     loop
         .
         .
        JDX  2A7
```

See also the EAX and LX1 POPs.

JPN Jump and Process Next item

0 J Affected: Control location; W0 or WORKLIST

-1 NJ

JPN is used to control a loop that steps through a list, from top to bottom, looking at each entry. Since different lists have different entry sizes, JPN must look at the Code word for the list to determine the increment value. (Actually JPN is coded in POPs and uses the PNI POP to make this determination.) Typically, the loop begins with a TPT POP, which creates a pointer to the top of a list and puts that pointer in W0. JPN increments the pointer in W0, jumps to "location" e if there is still a list entry remaining, or falls through (and discards W0) if there are no more entries. Thus JPN is equivalent to:

PNI WO

JAT e

MMO WORKBOT

JUMP related POPs:	Category:	Category Number:
SKR Reduce and Skip if negative	Memory Modification	4

SWE Set if WO Equal to memory, discard if equal

+1 T Affected: Answer (T if equal); WORKLIST

0 F

If WO equals the contents of e, remove WO and answer TRUE. If not equal, leave the WORKLIST alone and return FALSE.

SED Set if Equal and Discard

-1 Affected: Answer (T if equal); WORKLIST

If WO equals the contents of e, answer TRUE; otherwise answer FALSE. In either case, discard WO. This is the same as SWE except when the values are not equal.

SEK Set if Equal and Keep

0 Affected: Answer (T if equal)

If WO equals the contents of e, answer TRUE; otherwise FALSE. In either case do not remove WO. This is the same as SWE except when the values are equal.

SME Set on Masked Equality

0 Affected: Answer (T if equal in specified bit positions)

If WO equals the contents of e, in those bit positions specified by a mask in the "A" register, answer TRUE; otherwise answer FALSE. Do not remove WO. This is the same as SEK except for the masking. SME:SEK = (Sigma) CS:CW.

SWG Set if WO Greater

0 Affected: Answer (T if greater)

If WO is greater than the contents of e, answer TRUE; otherwise FALSE. Do not remove WO. Both values are treated as signed integers.

SGD Set if Greater and Discard

-1 Affected: Answer (T if greater); WORKLIST

If WO is greater than the contents of e, return TRUE; otherwise FALSE. In either case, do discard WO. This is the same as SWG except that WO is removed.

SWA Set if WO .AND. memory is non-zero

0 Affected: Answer (T if any bit positions are 1 in both values).

If the logical AND of WO and the contents of e is non-zero (i.e. there were corresponding 1's in any bit position), return answer TRUE; otherwise FALSE. Do not remove WO. SWA is typically used to test for the presence of a particular bit in WO. However, it should not be used to do the job of, say, SOF or SOP. Note the derivation of the mnemonic SWA from the 9-Series SKA instruction.

QMN Query Memory Negative

0 Affected: Answer (T if negative)

If the contents of e are negative, return answer TRUE; otherwise FALSE. See also TMT.

TMT Test Memory TRUE

0 Affected: Answer (T if memory TRUE)

If location e contains a .TRUE. value (sign bit set), return answer TRUE; otherwise FALSE. As implemented, this is the same POP as QMN, but they should not be used interchangeably (though occasionally they are). Compiler code should not take undue advantage of the knowledge that .TRUE. is the same as negative.

QZR Query ZeRo alternate mnemonic: TZR

0 Affected: Answer (T if memory zero)

If location e contains zero, return answer TRUE; otherwise FALSE. The mnemonic TZR (Test if Zero) is a holdover from the 9300 compiler.

SRC Set on Range of CENTRAL

0 Affected: Answer (T if INTCENT within range)

If INTCENT (Integer CENTRAL, which is CENTRAL2) is within the inclusive range specified by the two-word item beginning at location e, return answer TRUE; otherwise FALSE. Location e contains the lower bound, e+1 the upper bound. All values are treated as signed integers. Range specifiers need not begin at double-word boundaries.

SOR Set On Range

0 Affected: Answer (T if within range or untestable)

SOR is similar to SRC. It tests for a value within a range specified in locations e and e+1. The value is not in INTCENT, however. SOR expects a pointer in W0. If the pointer is to the ICONSLST (Integer Constant List), the range test is made on the value of the constant pointed to, with answer TRUE if the constant is within range; otherwise FALSE. If the pointer is not to the ICONSLST, then its value cannot be tested or is not worth testing at compile time, so SOR must return answer TRUE since it does not know that the value is out of range.

This POP is used to check the values used in statements like

```
REWIND 23
```

I/O unit numbers must be in the range 0-9999, but if the unit number is not specified with a constant, then it cannot be checked at compile time. For example:

```
REWIND N
```

SNE Set if list Not Empty

0 Affected: Answer (T if bottom file has something on it)

If the bottom file of list e is non-empty (i.e. there is something on it), return answer TRUE; otherwise FALSE. The list is not disturbed.

SRV Set if list ReserVed

0 Affected: Answer (T if reserved)

If list e currently has any reserve markers on it (i.e. if its Start is not equal to its Base), return answer TRUE; otherwise FALSE. The list is not disturbed.

SOT Set on Test set on ID field equality

0 Affected: Answer (T if equal)

Compare the ID field (first eight bits) of the pointer in WO with the ID field of location e. If they are the same, return answer TRUE; otherwise FALSE.

SOT can be used to establish the identity of a list or a plex or various other kinds of pointers and fake pointers. It also tests for the class (scalar, array, etc.) of Symbol Table pointers. Section 1.12, "Spectrum of ID Numbers", shows the different values that can appear in the ID field of a pointer, what they signify, and what locationn should be SOTted for to test for each number. For example, to test for scalar, use SOT SCALAR; to test for ICONSLST, use SOT ICONSCOD; to test for ADDPLEX, use SOT ADDPLEX. The mnemonic SOT is a holdover from the 920 compiler and has no useful meaning whatsoever.

SOM Set On Mode

0 AFFECTED: Answer (T if same mode)

Compare the mode field (third hex digit) of the pointer in WO with the mode field of location e. If they are the same, return answer TRUE; otherwise FALSE. This can be used to test for same mode between two pointers, but more often it tests for a particular mode by addressing a constant in LISTDEFS, such as INTEGER or COMPLEX.

SOF Set On Flag trait

0 Affected: Answer (T if trait present)

Find the flag word associated with the pointer in WO and compare it with the trait bits specified in location e. If any of those bits are present (i.e. if the logical AND is non-zero), return answer TRUE; otherwise FALSE. Although SOF can test for more than one trait, it usually tests for only one by addressing a constant in LISTDEFS such as CONSTFLG, BLANKCOM, or DUMMY. An example of a multiple test is the constant ALOCATED, which can be any of five traits.

It is not altogether obvious where to find the flags word associated with a pointer. If it is a plex pointer, the flags are contained in the second word of the plex definition (the plex flag word). If it is an ordinary list pointer, the flags are in the Flags parameter word. If it is a symbol pointer, the flags are in the third word (YSYMBID) of the Symbol Table entry pointed to.

TEST related POPs all other POPs that can make tests): Category Category: Number:

MOF Move Off	Manipulation 1
MOR Move Off and Release	Manipulation 1
TOT Take Off Top	Manipulation 1
TOR Take Off top and Release	Manipulation 1
SKR Reduce and Skip	Memory Modification 4
QBL Branch and Leave interpretive (BRL)	Jump 5
QJS Jump Recursively to Subroutine	Jump 5
TRY TRY	Jump 5
JDX Jump and Decrement indeX	Jump 5
JPN Jump and Process Next item	Jump 5
SER SEaRch	General List 7
SERK SEaRch with Key	General List 7
SOP Set On Permissibility	Specific List 8
SOL Set On Label	Specific List 8
TPT Top PoinTer	Pointer 9
PNI Process Next Item	Pointer 9
QIX Query Indicator to indeX	Pointer 9
CSA Character Scan with Answer	Input Scan 13
CSK Character Scan and Keep	Input Scan 13
SOC Set On Character class	Input Scan 13
QSA Quote Scan with Answer	Input Scan 13

1.7.2.7. GENERAL LIST

ZAP ZAP

O Affected: List e

Empty list e completely, removing all files. In other words, adjust the Bottom, Top, and Start to coincide with the Base.

EMP EMPty

O Affected: List e

Empty the bottom file of list e. This does not remove the file marker (if any). In other words, adjust the Bottom and Top to coincide with the Start.

CPY CoPY

O Affected: List specified in PO; PARAMLST

Copy the bottom file of list e onto the bottom of the list specified in PO. This does not affect the data on list e, nor does it copy the reserve marker (if any). A CPY is normally preceded by an ADR, to set up the list that is to receive the information. Afterwards, CPY automatically removes PO.

CAE Copy And Empty

O Affected: List specified in PO; List e; PARAMLST

Same as CPY, but also empties the bottom file of list e (but does not remove or copy the reserve marker). CAE:CPY = MON:MOK. In other words, CAE e is the same as CPY e, EMP e. CAE also removes PO.

CAR Copy And Release

O Affected: List specified in PO; List e; PARAMLST

Copy the bottom file of list e to the list specified in PO, then release list e (i.e. empty the bottom file and remove the reserve marker, if any -- see RLS POP, below). Since the reserve marker is removed, list e might not be empty after a CAR. As with CPY, a CAR is usually preceded by an ADR. Afterwards, PO is removed. CAR e is the same as CPY e, RLS e.

RSV ReSerVe

0 Affected: List e

Append a reserve marker to the bottom of list e. This creates a new, empty (i.e. Bottom = Top) file on list e. The structure of a reserve marker is described in Section 1.4.2, "How Lists are Stored".

RLS ReLeaSe

0 Affected: List e

Remove the bottom file, including reserve marker, from list e. This resets the Start and Top to the values contained in the two-word reserve marker. If list e has not been reserved, RLS acts just like EMP.

UNR UNReserve

0 Affected: List e

Remove the most recent file marker from list e, but without emptying the bottom file. Instead, combine the bottom file with the previous file. Since this involves sliding the bottom file by two words (into the space previously occupied by the reserve marker), pointers to information that was in the bottom file are no longer valid. If list e has not been reserved, UNR does nothing.

RAD Reserve and ADdress

0 Affected: List e; PARAMLST

Reserve list e and append its list number (e) to the PARAMLST. RAD e is the same as RSV e, ADR e, and is often used in preparation for a CPY or CAR. The two POPs RAD e1, CAR e2 can be used to copy the bottom file of e2 to e1, including the reserve marker.

CNT CouNT

+1 Affected: WORKLIST

Count the number of words in the bottom file of list e and append this count to the WORKLIST. The count is equivalent to Bottom - Top.

SER SEaRch

+1 T Affected: Answer (T if found); WORKLIST

0 F

Search list e for the item (possibly multi-word) contained in CENTRAL. If it is found, create a pointer to it, append this pointer to the WORKLIST, and return answer TRUE. If not found, leave the WORKLIST alone and return answer FALSE.

The size of item searched for depends on the list. Each list is defined with a certain entry size, key size, and offset (see Section 1.4.5, "Searching and Registering; Entry, key, and offset size"). These values are contained in the Code parameter. The SER POP uses the entry size to determine how to step from one entry to the next. It uses the key size to determine how many words to compare with CENTRL. And it uses the offset size to determine where, in the entry, to begin comparing. Thus, for example, SER automatically compares against the first two words of a six-word entry on the SYMTABLE. A key size of zero means one word, but only the lower 20 bits are compared. This is most often used for searching label lists for the label value.

For a one-word compare, CENTRAL2 is used. This is also called INTCENT (Integer Central) and is commonly used throughut the compiler for most single, integer values. For a two-word compare, SER uses CENTRLA1 and CENTRAL 2. For three-word, CENTRAL 1, 2, and 3. And for four-word CENTRAL 1, 2, 3, and 4. Thus the one-word compare is the odd one, since it uses CENTRAL2, (i.e. INTCENT) not CENTRAL1.

This list is searched from top to bottom, in the bottom file only. Items in previous file will not be found. If an item appears more than once in the bottom file, only the first (topmost) appearance will be found.

SERK SEaRch with Key

+1 T Affected: Answer (T if found); WORKLIST

0 F

SERK is the same as SER, except that instead of using the standard entry, key, and offset values found in list e's Code word, the user can specify his own (non-standard) values by loading a "fake" Code word into the A register. This word has the same format as a regular Code word, namely:

12 bits	12	4	4	4
ID+MODE		OFFSET	KEY	ENTRY

SERK uses the entry, key, and offset values from this word. In constructing the pointer to be returned, it uses the list's standard ID and mode fields from the list's Code word (unless the list is the SYMTABLE, in which case it must get the ID field from the YSYMBID word in the SYMTABLE entry). Thus, only the lower twelve bits of the parameter in the A register are used.

SORT SORT

0 Affected: List e

Sort the bottom file of list e into increasing order, top to bottom. Like SER, SORT uses the Code word of list e to determine the entry size (individual words are not sorted, the whole entry is moved intact), the key size (how many words need be compared in each entry to establish the sort sequence), and the offset (which word to begin the compare at). Values are treated as unsigned integers. (Thus X'FFFFFFFF' is the greatest value.) Obviously, pointers to list e are no longer valid after a SORT.

FLIP FLIP list upside-down

0 Affected: List e

Invert the bottom file of list e. This is done word by word, regardless of the entry size of the list. For example, on a list with one three-word entry, the bottom word becomes the top and vice-versa, while the middle word remains where it is.

LOD LOaD a list

0 Affected: List e

Append a string of words to list e. The EATEMP (see EAT POP) specifies the location of the string. The first word of the string is the count of the number of words following. It is the following words that are loaded onto the list. This POP was used primarily during debugging of the compiler, when it was useful to create "fake" information on lists and let parts of the compiler process that information. It is also used by "Output Print Buffer" (Section 2.3.4).

LODI LOaD list from Input

-1 Affected: List e; INPUTLST; WORKLIST

Like LOD, but gets the string of words from the scratch file (INPUTLST), and the count from WO, which it discards. This POP can only be used in passes 2 and 3, since Pass 1 gets its input from the SI file, not from a scratch file. If there are sufficient words on the INPUTST, LODI uses them and removes them from the top of the list (like the TOT POP). If there are not enough words, more will have to be read from the scratch file.

GENERAL LIST related POPs:	Category Category	Category Number:
MOR Move Off and Release	Manipulation	1
TOR Take Off top and Release	Manipulation	1
SNE Set if Not Empty	Test	6
SRV Set if ReserVed	Test	6
BOP BOttom Pointer	Pointer	9
TPT Top PoinTer	Pointer	9
PNI Process Next Item	Pointer	9
PUL PULl plex	Plex	11
RPL Reserve and PuLl	Plex	11

1.7.2.8 SPECIFIC LIST

(Symbol Table) (See also "Symbol Table POPs", Section 2.8.2.)

REG REGister

*0 Affected: W0; SYMTABLE; SYTCHLST

REG is used for setting up the traits (flags), mode, and/or class (ID field) of the Symbol Table item pointed to in W0. The third word (YSYMBID) of each SYMTABLE entry contains an eight-bit ID field (SCALAR, ARRAY, UNUSED, etc.), a four-bit mode field (INTEGER, UNKNOWN, etc.), and 20 bits of trait flags (DUMMY, BLANKCOM, etc.). REG can affect any or all of these.

Location e contains the same three fields (it is usually a constant in LISTDEFS). Typically, only one of the field is non-zero. The flag and mode fields are simply merged into the corresponding fields in the SYMTABLE word. The ID field is handled somewhat differently. If the new ID field (contained in e) is non-zero, it completely replaces the old ID field (in the SYMTABLE). This is because the initial (UNUSED) value of the ID field is not zero, but X'E0'. If the ID or mode fields of the SYMTABLE entry are changed, this change is also reflected in the pointer in W0. Since the pointer does not contain the flags field, it cannot reflect changes there.

Since REG may change values in the Symbol Table that later have to be restored (if a scan fails and a rescan must be made), all such changes are recorded on the SYTCHLST (Symbol Table Change List). For further information on this, see "Saving and Restoring Information", Section 2.7.1.

SUP Set Up Permissibility

0 Affected: SYMTABLE; SYTCHLST

SUP performs an extract (logical AND) between the contents of e+2 (the permissibility word) and the fifth word (YPERMWRD) of the Symbol Table entry pointed to in W0, and stores the result back into the SYMTABLE permissibility word. This is used to remove the permissibility bits of usages that are no longer legal. As with REG, changes to the SYMTABLE are recorded on the SYTCHLST. For further details, see Section 2.8.1, "Permissibility".

SOP Set On Permissibility

0 Affected: Answer (T if permissible)

If the AND of location e+1 (the SOP word) and the fifth word (YPERMWRD) of the Symbol Table entry pointed to by WO is non-zero (i.e. if that permissibility bit is present), return answer TRUE; otherwise FALSE. This is used to test the permissibility set up by SUP. It does not change the SYMTABLE and therefore does not need to record anything on the SYTCHLST.

(Label Lists)

SUL Set Up Label

0 Affected: List pointed to in WO; SYTCHLST

SUL is used to register traits (e.g. DEFINED) on an entry on a label list (e.g. the LOCLBLST). These traits are ketp in the upper 12 bits of the first word of a label list entry (YSULWORD). SUL merges the contents of e (the SUL word) into the label list word. These traits are tested by the SOL POP. As with REG, the changes made to the label list must be recorded on the SYTCHLST (even though this is not the Symbol Table).

SOL Set On Label

0 Affected: Answer (T if trait present)

SOL tests the label traits set up by SUL. If the AND of the contents of e (the SOL word) and the first word (YSOLWORD) of the label list entry pointed to by WO is non-zero (i.e. the trait is present), return answer TRUE; otherwise FALSE.

(Integer Constant List)

RGI ReGister Integer constant

+1 Affected: ICONSLST; WORKLIST

Register the contents of memory location e on the ICONSLST. This means if the value is already on the ICONSLST, a pointer to that value is returned on the WORKLIST. If the contents of e are not previously registered on the ICONSLST, the value is added to the bottom, and a pointer to the new entry is returned.

Thus, RGI N is equivalent to

```
LDA  N

STA  INTCENT
```

JRS. RGINTCON

RGI recurses into POP language to accomplish this.

(Error List) (See also "Delayed Error Messages", Section 2.6.3.)

ERC ERror on Current character

0 Affected: ERRORLST; TRYLIST

ERC records (on the ERRORLST) an error at the current character position. This is done with a two-word entry consisting of the character count and location e, (the location of the error message). If the error is a fail level error, ERC also sets the Delayed Fail Flag on the TRYLIST.

ERL ERror on Last active character

0 Affected: ERRORLST; TRYLIST

Same as ERC, but uses the character position of the last active character instead of the current character. This enables the compiler to print a marker on the character immediately preceding the current character.

ERW ERror on character specified in WO

-1 Affected: ERRORLST; TRYLIST; WORKLIST

Same as ERC, but uses the character count specified in WO instead of the current character count. ERW then discards the value in WO. If the compiler can anticipate that later on an error marker may be needed at the current position, it can save the character count somewhere, then FET it and use ERW. This may even back up to a previous continuation card to attach the error marker, since none of the cards are printed until the whole construct has been scanned.

(Save List) (See also "Saving and Restoring Information", Section 2.7.1.)

SAL Save A List

0 Affected: SAVELIST

SAL is used to remember the state of a list by saving its Bottom, Top, and Start (relative to the Base) on the SAVELIST, along with the list number. Actually, not all of those values are saved if they do not differ from one another. See Section 2.7.1.

SAV Save A Value

0 Affected: SAVELIST

Save the contents of e by appending its value to the bottom of the SAVELIST. This technique is used only in Pass 2; Pass 1 uses SAL, and a subroutine (SAVEITEM) for saving values. The location of the saved value is not recorded. The compiler must keep track of restoring (with RST) values to the right places.

RST ReSTore a value

0 Affected: Memory location e; SAVELIST

Restore the contents of location e by storing the bottom word of the SAVELIST there. Then remove that bottom word. RST is used in conjunction with SAV.

(Code List) (See also "CODELIST Manipulation in Pass 1", Section 2.12.)

FIL FILe on Code List

0 Affected: CODELIST

Append the contents of location e to the bottom of the CODELIST. Used by Passes 1 and 2 to collect interface items, which will later be moved to the Output List and possibly to a scratch file.

FAD File ADdress on Code List

0 Affected: CODELIST

Same as FIL, but files the immediate value of e rather than the contents of location e. This is used particularly by Pass 1 to file the constant values identifying statement types. FAD N is equivalent to FIL =N.

(Group List)

COG COpy Group

+1 Affected: GROUPLST; WORKLIST; list e

Groups are variable-sized data structures much like plexes. The reason for having groups is that the Plex List generally gets wiped out after

every statement, whereas the information contained in groups is needed throughout the compilation. The GROUPLST is never destroyed. The data kept in groups generally has to do with array dimensions.

COG builds a group from the bottom file of the addressed list e and appends it to the GROUPLST. A group consists of a count word n indicating that n words of data follow (n may be zero), and then the n data words themselves. A pointer to the group is appended to the bottom of the WORKLIST. The bottom file of list e is emptied, but the reserve marker, if any, is not removed. Thus

```
COG  e
```

is equivalent to the sequence

```
BOP  GROUPLST
CNT  e
MON  GROUPLST
ADR  GROUPLST
CAE  e
```

and that is how COG is done (It is coded in POPs to save space.)

PUG PUll Group

-1 Affected: List e; WORKLIST

PUG is to COG as PUL is to the plex-building POPs. Assuming WO contains a pointer to a group* on the Group List (GROUPLST), PUG copies the items (not including the count) from the group onto the bottom of the addressed list e. WO is removed. No change is made to the GROUPLST.

* Actually, the pointer points to the count word of the group.

SPECIFIC LIST related POPs:		Category Category:	Number:
LODI	LOaD list from Input	General List	7
PUL	PULl plex to a List	Plex	11
RPL	Reserve and Pull to List	Plex	11
CIC	Copy Inherit and Count	Plex	11
CIF	Copy Inheriting First-term traits	Plex	11
FIC	Fixed-length plex Inherit first-term traits	Plex	11
FIP	Fixed-length plex Inherit from all traits	Plex	11

1.7.2.9. POINTER

BOP BOttom Pointer

+1 Affected: WORKLIST

BOP creates a pointer to what would become the bottom of the addressed list e if one more item were added to it (i.e., a pointer to the item about to be created). The pointer is appended to the bottom of the WORKLIST.

TPT Top PoinTer

+1 T Affected: Answer (T if list not empty; F if empty); WORKLIST

0 F

TPT is similar to BOP, but creates a pointer to the top word in a list. The Answer Flag is returned TRUE if there is anything in the bottom file of the list; otherwise, the Answer Flag is returned FALSE and no pointer is created. The pointer (if any) is appended to the bottom of the WORKLIST.

FRP Fetch Relative to Pointer

+1 Affected WORKLIST

This POP computes the effective address of the pointer in WO, then adds to this the effective address e (which is usually a small absolute value, called a Rung Number, and quite often zero). This resultant address is used to retrieve a word from memory, and this word is appended to the bottom of the WORKLIST.

GRP Get Relative to Pointer

*0 Affected: WO

This POP is just like FRP except that it replaces WO (the pointer) with the fetched word instead of appending it to the bottom of the WORKLIST.

BNG BriNG relative to pointer (alternate mnemonic for GRP)

*0 Affected: WO

POX POinter to indeX

0 Affected: "X1" register

This POP addresses a pointer in memory location e. The effective address pointed to by the pointer is placed in Index 1. Index 1 is then used to modify the address(es) of following POP(s). Traditionally, Index 1 could be used only to modify the address of the POP following the POX. (On the 9300 this was a requirement, since the Interpreter destroyed X1 on every POP execution.) On the Sigma, X1 is not destroyed, but care should be taken that no POPs are executed that could cause a reassignment of memory. Such a reassignment would invalidate the address in X1. If you write a POP that <u>does</u> cause a reassignment, then do the POX over again before using X1.

PNI Process Next Item

*0 Affected: Answer (T if new pointer does not point beyond end of list; F if it does); W0

PNI is used in conjunction with TPT. It increments the pointer in W0 by the entry size (item size) of the list e so that the pointer now points to the next entry. If there <u>is</u> another entry, the Answer Flag is returned TRUE; otherwise, the Answer Flag is returned FALSE, in which case the pointer in W0 points beyond the bottom of the list e.

QIX Query ID field and copy to indeX

0 Affected: Answer (TRUE if in range, FALSE if out of range); X1

This is a peculiar POP. QIX copies the ID field (bits 0 thru 7) of the contents of W0 into X1, right-justified with leading zeroes. It then sets the Answer Flag TRUE if the ID value lies in the range specified by the effective address e (where the lower bound is given in bits 16 through 23 and the upper bound in bits 24 through 31). The Answer Flag is set FALSE if the ID field is not within the given range.

QIX is assembled by a special system Proc (in SYSTEM FORTCOMP) which expects as its argument a symbol that is defined by the RANGE FUNCtion. The latter is defined in the assemblies that need it, which happen to be PASS2 and OUTPROG. One might write, for instance,

```
HEXRANGE  EQU  RANGE(0,15)

          QIX  HEXRANGE
```

which puts the indicator (bits 0 to 7 of W0) into X1 and then tests whether that value lies between 0 and 15 (inclusively). The RANGE FUNCtion yields a three-element list as its value. The elements are the two arguments given, followed by a zero to give a third list element. The reason we want a third element is that such special operands as W0 and P2

are defined as two-element lists, and we wanted to be able to distinguish ARANGEs from WO and the like. This merely enables us to give better diagnostics at the time the compiler is assembled.

The LOW FUNCtion (also defined where it is needed--i.e., OUTPROG and PASS2) is used to retrieve the lower bound of a range. Thus LOW(HEXRANGE) is 0. LOW is useful with the DXI POP.

The QIX POP is used in preparing for an indexed branch into a table, based on the identifier portion of a plex pointer or the indicator portion of a code word in the Pass2/Pass3 interface format. A typical example of the use of QIX and DXI is shown below. It is taken from OUTPROG in Pass 3.

```
OUTRANGE  EQU  RANGE(P23HABS,P23RPROG)  HOLLERITH MULTIPLE-

*                                       WORD ABS THRU RETURN-TO-PROGRAM

*                                       LOCATION INDICATOR

          ...

8Z1  JRS  NXINWORD          GET OPERATOR

     QIX  OUTRANGE          IS ID FIELD IN RANGE ?

     JRS,F OPABORT          NO. BEGONE.

     DXI  LOW(OUTRANGE)         YES.

     JRS  OUTJUMPT,X1           CALL APPROPRIATE OUTPUT ROUTINE
```

The symbols P23HABS and P23RPROG are external and are defined in LISTDEFS. This is also true of plex numbers, which is what are used in Pass 2 to control jump tables. OUTJUMPT is a table of BRU POPs.

POINTER related POPs:	Category:	Category Number
JPN Jump and Process Next item	Jump	5
SOR Set On Range	Test	6
SOT Set On Test (Set on WO ID field equality)	Test	6
SOM Set On Mode equality with WO	Test	6
SOF Set On Flag (trait)	Test	6
REG REGister	Specific List	8

SUP Set Up Permissibility	Specific List	8
SOP Set on Permissibility	Specific List	8
SUL Set Up Label	Specific List	8
SOL Set On Label	Specific List	8
PUG PUll Group	Specific List	8
PUL PULl a Plex	Plex	11
RPL Reserve and PuLl	Plex	11

1.7.2.10 ADDRESSING

ADR ADdRess to Parameter List

0 Affected: PARAMLST

This POP might well have been called EAP (Effective Address to PARAMLST) since it is quite analogous to EAT and EAX, discussed below. The effective address e (with zero in bits 0-14) is appended to the bottom of the PARAMLST. Thus ADR has an immediate address. It is quite useful for passing arguments to subroutines, since it is easy to make indirect references through the bottom few items on the PARAMLST. (P0, P1, P2, etc.).

In addition, certain other POPs expect an extra argument in P0. These are CAE, CAR, CPY; all of them discard ("ditch") P0.

EAT Effective Address to Temp EATEMP

0 Affected: EATEMP

The effective address e replaces EATEMP 15-31, and zero replaces EATEMP 0-14. Thus EAT has an immediate address and is quite similar to EAX. Quite a few subroutines expect an argument in EATEMP. In addition, several POPs expect an extra argument in EATEMP. These are FTE, LOD, PAC, PAM, PDC, PHC.

EAX Effective Address to indeX

0 Affected: "X1" register

The effective address e of the POP is placed in X1, with the upper bits zero. In other words, EAX has an immedite operand.

DXI Decrement indeX Immediate

0 Affected: "X1" register

The effective address e is subtracted from the contents of X1. This is handy for getting around the 16-bit Sigma POP address field where an

operand of "-1,X1" will not have an expected effect since the -1 will be truncated to sixteen bits and will have the resultant value 65535.

If one wanted to get at "T1", the first term up from the bottom of the TERMLIST, it would be tempting to write

```
LX1   TERMBOT

FET   -1,X1
```

but it would not work. What you have to write instead is

```
LX1   TERMBOT

DXI   1

FET   0,X1
```

And that is the main reason we have the DXI POP.

FEX Fail EXit

0 Affected: memory location called FAILEXIT

The effective address e (with zero in bits 0-14) is stored in the temp known as FAILEXIT. The FEX POP tells where to go when a Fail occurs, provided that the Fail was not under control of a TRY POP. See the section on TRY-Fail (Section 2.7).

WAD WORKLIST Address Direct

0 Affected: the POP being executed

Nobody ever writes a WAD POP directly. WAD is generated automatically by the assembler (Meta-Symbol) when another POP addresses W1, W2, and so forth. (For W0, the assembler generates an indirect reference to WORKBOT.) Bits 16 through 19 of the effective address e are subtracted from (WORKBOT) to determine a new effective address. Bits 20 through 31 tell which POP to execute with the new effective address. Thus this POP seems backward: the op code is in the address field, in a way, and the address is in the op code. It is peculiar; but it works just fine.

PAD Parameter Address Direct

0 Affected: the POP being executed

Nobody ever writes a PAD POP directly. PAD is generated automatically by the assembler when another POP addresses P0, P1, P2, and so forth. PAD is

just like WAD except that it operates through the PARAMBOT instead of the WORKBOT. PAD is used less frequently than PAI.

PAI Parameter Address Indirect

0 Affected: the POP being executed

Nobody ever writes PAI POP directly either. PAI is generated automatically by the assembler when another POP references POIND, P1IND, *P0, *P1, and so forth. Bits 16 through 19 of e are subtracted from (PARAMBOT) to determine a new effective address f; then bits 16 thru 19 of location f become the effective address of the POP whose op code is specified in bits 20 to 31 of the PAI POP.

ADDRESSING related POPs:	Category Category:	Category Number:
LX1 Load indeX 1	Manipulation	1
RAD Reserve And Address	General List	7
POX POinter to indeX	Pointer	9
QIX Query ID field to indeX	Pointer	9

1.7.2.11 PLEX

FIP Fixed-length plex, Inherit from all terms

-n+1 Affected: WORKLIST, PLEXLIST

This is the most popular plex-building POP to use when you know how many terms are going to be in your plex. FIP builds a plex from the bottom n items on the WORKLIST where n = 1, 2,..., (A larger value of n would work, but nobody happens to use it.) The n is suppied in the two-word plex description addressed by FIP. This description doubleword is located in the assembly LISTDEFS. Traits are inherited from the merge of all the traits of the items in the plex (see Inheritance of Traits, Section 1.13.1). The n items are removed from the WORKLIST and a pointer to the plex is appended.

FIC Fixed-length plex, Inherit first-term traits

-n+1 Affected: WORKLIST, PLEXLIST

This POP is a minor variation on FIP. It is just the same except that thait inheritance works only on the first term (the farthest from the bottom) instead of all the terms. It is useful for such plexes as REPLPLEX (Replacement Plex, used with =) and FUNCPLEX (Function Call Plex).

PLO PLex Open

0 Affected: PLOWORDS (a 2-word temp)

PLO is used in connection with CIC and CIF to build a plex of variable size. PLO addresses a two-word plex description constant which is placed in PLOWORDS and PLOWORDS+1.

CIC Copy, Inherit and Count

+1 Affected: List e; WORKLIST; PLEXLIST

CIC (analogous to FIP) is used in connection with PLO to build a variable-sized plex. The bottom file of the addressed list e is built into a plex. Traits are inherited from all the terms of the plex. The list e is emptied, but the file mark is not removed. The plex pointer is appended to the bottom of the WORKLIST.

CIF Copy and Inherit First-term traits

+1 Affected: List e, WORKLIST, PLEXLIST

CIF (analogous to FIC) is like CIC except that inheritance is done only from the first (top-most) word instead of all the words.

PUL PULl a plex

-1 Affected: List e, WORKLIST

WO contains a pointer to a plex. PUL removes WO, then copies the items of the plex to the bottom of the addressed list e. (The bottom file of e is generally empty when this is done.) PUL may address the WORKLIST.

RPL Reserve and PuLl a plex

-1 Affected: List e, WORKLIST

RPL is just like PUL except that it places a reserve mark on the list e before pulling the terms. RPL WORKLIST will not work in any useful way.

1.7.2.12. PRINT

More detailed information on these POPs may be found in "Print POPs", Section 1.14.

PRC PRint Character

0 Affected: PRBUFFER; PCHPOS

The right-justified EBCDIC character specified by e (not the contents of e) is inserted into the print buffer at the position given by PCHPOS (Print Character Position). PCHPOS is then incremented by 1.

PRQ PRint Quote

0 Affected: PRBUFFER; PCHPOS

The quote (text string enclosed in dollar signs) beginning at location e is inserted into the print buffer. See also "Quotes", Section 1.15.

PAC Print Alphanumeric with Count

0 Affected: PRBUFFER; PCHPOS

The character string (not a quote) beginning at location e is inserted into the print buffer. The number of characters is determined by the count value in EATEMP.

PAM Print Alphanumeric, Minimum width

0 Affected: PRBUFFER; PCHPOS

Same as PAC except that trailing blanks are not printed, so the number of characters inserted into the print buffer may be less than the count specified in EATEMP. It can never be more; EATEMP specifies the maximum number of characters.

POC Print Decimal with Count

0 Affected: PRBUFFER; PCHPOS

The contents of e are converted to decimal and inserted into the print buffer, right-justified in a field whose width is determined by the count in EATEMP. This works much like an In format at run-time.

PDM Print Decimal, Minimum width

O Affected: PRBUFFER; PCHPOS

Same as PDC, except no field width count is specified. The value uses as many character positions as are required to express it. This works much like widthless I format at run-time.

PHC Print Hexadecimal with Count

O Affected: PRBUFFER; PCHPOS

The contents of e are converted to hexadecimal and inserted into the print buffer, right-justified (with leading zeros) in a field whose width is determined by the count in EATEMP. This works much like a Zn format at run-time.

PBL Print BLanks

O Affected: PCHPOS

The Print Character Position is incremented by the immediate value of e. This can move right or left, and does not actually print blanks. It works much like an nX format at run-time.

TAB TAB to print column

O Affected: PCHPOS

The print position is set to the value of e. TAB cannot be used to back up.

PRL PRint Line s

O Affected: PCHPOS; LO output file

The current print line is output on the LO device, followed by e-1 blank lines, if e 1. However, if the operand e is PAGE (defined as a negative value), no printing is done. Instead, a page eject is given.

GNL Guarantee N Lines on LO page

O Affected: LO output file

N is the immediate value of e. If there are fewer than N lines remaining on the current LO page, GNL gives a page eject. If there are at least N

lines remaining, or if the compiler cannot determine how many lines remain, GNL does nothing. This is used to assure that groups of lines get printed on the same page.

PRINT related POPs:		Category:	Category Number:
CNL	Comment Next Line	Generating (Build)	14

1.7.2.13. INPUT SCAN

Further information on these POPs may be found in "Character Scanning", Section 2.2, and "Character Translate Table", Section 2.4.

CSA Character Scan with Answer

0 Affected: Answer (T if character found); CURRCHAR; INPCHCT;

LACTCHCT; INPUTLST; SI input file; ALPHALST; SO output file

Test if the next source character is the one specified by the (right-justified) immediate value of e. If so, answer TRUE and step past the character. If not, answer FALSE and leave scan position alone.

If CURRCHAR (Current Character) is non-zero, CSA compares its value with the full-word value obtained from position e of the CHTRATBL (Character Translate Table). If unequal, CSA exits FALSE. If equal, it zeroes CURRCHAR and exits TRUE.

IF CURRCHAR is zero, the next input character must be fetched. This is handled by RDYSCNCH (Ready Scan Character) and may involve reading a new source card (which may alter SI and SO) and appending it to the INPUTLST. Then CSA proceeds as described above.

CSK Character Scan and Keep

0 Affected: Answer (T if character found); CURRCHAR; INPCHCT;

LACTCHCT; INPUTLST; SI input file; ALPHALST; SO outut file

Same as CSA, except that CSK does not step past the character. It just tests it for equality to e and leaves the scan position alone so that the character can be rescanned.

Although CSK does not step past the current character, note that if CURRCHAR is zero, a new input character will have to be fetched, which, like CSA, may involve reading from SI.

CSF Character Scan or Fail

0 Affected: Control location; CURRCHAR; INPCHCT;

LACTCHCT; INPUTLST; SI input file; ALPHALST; SO output file

Same as CSA, except that if the specified character is not found, CSF transfers control to SYNFAIL (Syntax Fail), instead of returning an

answer. As with CSA, if the character is found, CSF steps past it (by zeroing CURRCHAR). CSF is used when there are no more alternatives for an input character, and is equivalent to a CSA followed by a JAF to SYNFAIL.

SOC Set On Character class

O Affected: Answer (T if class matches); CURRCHAR; INPCHCT;

LACTCHCT; INPUTLST; SI input file; ALPHALST; SO output file

Test if the next source character is one that falls into any of the classes specified by the bits in e. If so, return TRUE; otherwise FALSE. Do not step past the current character in either case.

The classes than can be specified include such things as: letter; digit; IJKLMN (implicit integer); and the characters that can begin a "quote"-type statement (e.g. READ begins with R but no statement begins with J). Each SOC POP has, as its immediate address, a "flag" word (defined in FORTTEMP, along with the Character Translate Table) containing bits corresponding to the classes to be tested for. These bits are compared against the bits in the lower half of CURRCHAR, which was taken from the Character Translate Table. The value for each character in the CHTRATBL contains the bits for each class that it belongs in. For example, the character "4" contains both the "digit" and "hex digit" bits. The LETDIGFG (letter digit flag) contains both the "letter" and "digit" bits. Therefore, a SOC LETDIGFG would succeed on the character "4" because the "digit" bits matched. The same SOC on the character "+" would not succeed, because the CHTRATBL word for "+" contains neither the "letter" or "digit" bits.

Like CSK, SOC does not zero CURRCHAR, but it may have to read from SI in order to get something into CURRCHAR.

QSA Quote Scan with Answer

O Affected: Answer (T if quote found); CURRCHAR; INPCHCT; LACTCHCT;

INPUTLST; SI input file; ALPHALST; SO output file

Test whether the next n active source characters exactly match those in the quote (character string enclosed in dollar signs) beginning at location e. The number of characters compared (n) is determined by the length of the quote. If the entire quote is found, step past those characters and return answer TRUE. If not, reset to the former scan position (as if the POP had never been executed) and return answer FALSE.

QSA is much like CSA, except that it addresses a variable length string of characters. (See also "Quotes", Section 1.15.)

1.7.2.14. GENERATING Build)

BIN Build Instruction with Normal address

-1 Affected: CODELIST; WORKLIST

This is a very basic code-generation POP and works in a manner similar to several other code-generation POPs, namely, BIF, BAF, and BIM and, to a lesser extent, BINR, BIFR, BAFR and BIMR. The effective address e is the location of a word containing the op-code to be generated, plus a lot of supporting information. This word goes onto the CODELIST first. WO contains the operand pointer (e.g., to a scalar or to a statement label), and this goes onto the CODELIST second and is discarded from the WORKLIST.

The supporting information in the op-code word at location e is of little importance to the BIN POP but is useful to Pass 3 and also to the BIF POP, which is a more sophisticated version of the BIN POP. BIN cannot handle indirect addressing or subscripting and hence cannot handle dummies or arrays.

The most obvious supporting information contained in location e are the Indirect Addressing bit, the R Field and the X field of the instruction to be generated. Less obvious information tells Pass 3 how to list the instruction. (For example, LOGL, the calling-sequence op-code for LOGICAL, has the same bit configuration as AI, the Add Immediate instruction--X'20'. Also, for most instructions the R field is printed as an unsigned integer, but for MTW it is printed signed.) Then there is the all-important Index-Multiplier field which, although totally unimportant to the BIN POP, has quite an effect on BIF. This number (of value 1 or 2) tells by what amount the contents of an Index Register will be effectively multiplied if this instruction happens to be indexed, i.e., for a Word or Doubleword instruction.*

BINR BIN with RTEMP register field

-1 Affected: CODELIST; WORKLIST

BINR is just the same as BIN except that the R field of the generated instruction comes from RTEMP instead of memory location e. BINR and all the other generating POPs with a trailing R are for generating instructions dealing with expressions of mode INTEGER, which are handled in index registers if they happen to be reasonably simple subscripts, and are handled in Register 9 if they are complicated subscripts or not subscripts at all. RTEMP is initialized to 9 at the beginning of Pass 2 and altered whenever index register code comes up.

* A zero value in the Index Multiplier field indicates an immediate instruction, which is important to OUTPROG in Pass 3. From this it determines whether to produce a 20-bit or 17-bit address.

BIF Build Instruction with Full addressing capability

-1 Affected: CODELIST; WORKLIST; XREGLIST; DREGLIST; SCRPTLIST

Superficially, BIF and BIN are the same (so see BIN for a description of the operands), but BIF goes into POP language and JRSs to a routine called ADRCALC that handles the Addressing Problem in all its complexity. Thus for BIF, WO may be subscripted, or a dummy, or both, and BIF will automatically alter the instruction being generated (to include Index and Indirect fields) and possibly precede the requested instruction with other instructions to evaluate the subscript that is required. Thus a single BIF may cause many instructions to be generated.

For a full discussion of ADRCALC, see "Address Calculation", Section 2.14.3.1. See also "Retroactive Subscript Generation", 2.14.4.

BIFR BIF with RTEMP register field

-1 Affected: CODELIST; WORKLIST; XREGLIST; DREGLIST; SCRPTLST

BIFR is just the same as BIF except that the R field of the generated instruction comes from RTEMP instead of location e. See BINR.

BAF Build Absolute instruction and File

-1 Affected: CODELIST; WORKLIST

BAF expects a value in WO that is to become the address portion of the instruction that is to be generated. The op code is in the word addressed by e in the same for as for BIN and BIF. BAF truncates the WO value to 24 bits and tacks on an indicator that Pass 3 is to treat this as an absolute value. (Pass 3 will truncate this value further--to 20 bits or 17 bits as the instruction demands.) BAF can be used to generate immediate instructions, and it used to generate the LI,14 instruction that transmits the argument count in a standard subprogram call. Immediate instructions are also sometimes built with the BIM POP (or BIMR POP), as explained later under BIM.

BAFR BAF with RTEMP register field

-1 Affected: CODELIST; WORKLIST

BAFR is just the same as BAF except that the R field of the generated instruction comes from RTEMP instead of from memory location e. See BINR.

BIM Build IMmediate instruction

-1 Affected: CODELIST; WORKLIST

The name of this POP is very misleading, because immediate instructions can be built with BIN, BIF, or BAF as well as BIM. What BIM really means is "build an immediate instruction and observe special conventions during Pass 3". The operand in WO is expected to be a pointer to one of the following four constant lists: ICONSLST, HCONSLST, XCONSLST, or RDCONLST. The pointer in WO is left unchanged and is sent to Pass 3 as it is. But the left-most 8 bits of the op-code word at location e are changed to P23IMMOP" (from "P23OPCO") so that Pass 3 will know that it is to do something special.

The usual thing Pass 3 does when it receives a pointer to one of the constant lists as an operand is to create a literal. For example, if I have an ICONSLST pointer in WO, representing the value 3, and I BIF an AI,9 instruction I will get

```
AI,9  =3
```

If, however, I BIM an AI,9 instruction, I will get

```
AI,9  3   .
```

Because of the BIM, I get the value of the constant instead of the location of the constant.

The only reason for desiring this rather strange convention is that the pointer to the constant list can control the way in which the constant is listed. For example, the value zero may be listed as 0 or 0.0 or .FALSE. depending on its mode and whether it is on the ICONSLST or the RDCONLST. It is the same constant (and would be the same literal, if a literal were generated), but the listing shows the intention of the constant and is often a rendering of the source form in which the user wrote the constant.

It is conventional to QJS to CHEKIMED (Can Operand Be Expressed As Immediate?) and then BIM an immediate op code if the answer is TRUE and BIF another op code if the answer is FALSE.

BIMR BIM with RTEMP register field

-1 Affected: CODELIST; WORKLIST

BIMR is just the same as BIM except that the R field of the generated instruction comes from RTEMP instead of memory location e. See BINR.

BLC Build Library Call

0 Affected: CODELIST; DREGLIST; and ultimately REFLIST (during ASMCODE)

BLC builds a BAL,6 instruction to some routine in the library whose name is the two-word constant beginning at word location e. This constant is assumed to be available to both Pass 2 and Pass 3, since it is the address of the constant rather than its value that is sent to Pass 3. Thus BLC puts only one word on the CODELIST, whereas the other generating POPs so far mentioned put out two.

There is a third word associated with each library name, and it indicates which of the even-numbered index registers this routine destroys, if any. This information is stored into the DREGLIST. Actually, only Registers 2 and 4 are important since all routines that can be called with BLC destroy Register 6 by linking through it.

BRO Build with Register Operand

0 Affected: CODELIST

BRO builds an absolute register-to-register instruction. Like BIF, it gets the op code, indirect bit, index and register fields from the addressed word e. It also gets its address, as a register number, from this word, and it tacks on an Absolute Address indicator and puts this out as the operand word onto the CODELIST.

This POP is used to build such register-to-register instructions as

```
LW,4   9
```

and

```
FML,8  8   .
```

BRO does not disturb the WORKLIST.

BROR BRO with RTEMP register field

0 Affected: CODELIST

Same as BRO except that the register field is taken from RTEMP rather than the addressed word e. The address field still comes from e. This POP builds instructions involving a fixed operand register and a variable R register. BROR does not disturb the WORKLIST.

BRBR BRO with RTEMP register field and RTEMP address field

0 Affected: CODELIST

Same as BRO, except that both the register field and the address field are taken from RTEMP. This POP builds instructions involving a common variable address and register--i.e., register-to-same-register instructions.

CNL Comment Next Line

0 Affected: CODELIST

CNL is used for providing comments to be printed on the object code listing. It addresses a character string, and causes that string to be printed in the comment field of the next instruction to be generated. The address of the character string is passed, so the string must be available to both Pass 2 and Pass 3. If the LO option is not specified, CNL does nothing.

GENERATING related POPs:	Category Category:	Number:
BAM Build by Address Mode	Control	15

1.7.2.15 CONTROL

BAM Build by Address Mode actually execute by address mode

0 Affected: Nothing, except what may be affected by the executed POP.

Although this POP is usually used for code generation, it does not actually build anything. It takes the mode field (bits 8-11) of WO, shifts it to the address field, and adds it to the effective address e. It then executes the POP at the location addressed. Thus it should really be called EAM (Execute by Address Mode).

The reason we say "Address Mode", by the way, instead of "WO mode" is that WO usually contains the pointer that will become the address field of the generated instruction. The instruction addressed by BAM is often a BIF POP, which specifies the op code to be built. If it does address a BIF, of course, the size of the WORKLIST will be reduced by one.

XEQ Execute

0 Affected: Nothing, except what is affected by the executed POP

XEG executes the POP at location e. That POP may be another XEQ if desired. This POP is often used in connection with the PARAMLST, where an instruction to be executed is given as a subprogram argument. Using XEQ to execute a JRS can be quite powerful.

NUL NULL

0 Affected: Nothing

This POP does nothing at all, and it takes only 5.8 microseconds not to do it (on a Sigma 7). It is useful either as a "do nothing" entry in a table of instructions that are to be selectively executed (e.g. with a BAM POP) or as a "do nothing" location to ADR when entering a routine that expects to be given a POP to execute as one of its arguments.

1.7.2.16. DEBUG

DTN Debug Trap eNter

0 Affected: Typewriter control in CDEB mode

Causes the Interpreter/Debug Package to enter trap mode.

DTX Debug Trap eXit

0 Affected: Typewriter control in CDEB mode

Causes the Interpreter/Debug Package to exit from trap mode.

1.8 Recursion

Most of the coding in the compiler is recursive, which does not necessarily mean that it actually recurses, but it could. Note that this is true recursion capability, not just re-entrance or multiple user. It means that subroutines can call themselves, either directly or indirectly, and very little care is required to keep the levels separate. Large parts of the compiler do not need or use this capability, but it costs very little to provide it automatically, and it gives some other advantages.

1.8.1 POPs and Lists

The main thing that facilitates recursion is that most of the coding is in POPs and these POP do most of their processing on push-down type lists. Each subroutine can append information to the bottom of a list without disturbing information that was placed on that list by upper level routines. It then has the responsibility of removing the data it added, and only that data, so that the list on exit looks the same as it did on entry. Of course there are exceptions, where a subroutine is specifically designed to return with something added to a list, or to process and remove something given to it on a list. But for the most part, the guiding principle is to leave lists clean.

Subroutines often want to work with a list as if the whole list were available, appending variable amounts of data and then copying it all to another list, or building a plex out of it. The RSV (Reserve) and RLS (Release) POPs are used for this. Essentially, they create and remove file markers. Most of the other list operation POPs deal only with the bottom file of a list, thus giving the effect of having a whole clean list at each level of recursion.

1.8.2 Calling and Returning

The principal list involved in recursion is the EXITLIST. It serves two purposes. At each call, the return address is appended to the bottom of the EXITLIST. Then, upon returning, control is returned to that address plus one, and the address is removed from the EXITLIST. Thus, for example, at a call nesting of seven, there will be seven addresses on the EXITLIST. In addition, there is an answer flag for each level, and this is kept on the EXITLIST (in the first bit of the word containing the return address). This means that each level can do tests and create answers without disturbing the answers at other levels. It is safe to call a subroutine and know that it will preserve the current answer (unless, of course, the subroutine is specifically designed to return an answer). The following section illustrates a case where this can be used to preserve answers, even though a subroutine is not really needed.

It is difficult to pass the current answer down to a subroutine. This is not a procedure that the compiler uses. It would be equally difficult to pass answers back up to the calling level, except that a number of special exits are provided for this purpose. These are described in Section 1.9.3.3,

"Recursive Exits". The most common exit points are EXIT (return with no answer), EXITTRUE (return with answer TRUE), and EXITFALS (return with answer FALSE). These are POP language exits, reached with a BRU POP. It is normally recommended to use the equivalent machine language exits, EXITML, EXITTRML, and EXITFAML, reached with a BRL POP.

The principal POP involved in recursion is the JRS POP (Jump Recursively to Subroutine). This is used to call any POP language subroutine. There is an alternate mnemonic, QJS (Query and Jump to Subroutine) that may be used to indicate that the subroutine is going to return an answer. This is a fairly recent convention, and is not consistently followed in the compiler. (The same is true of the alternate mnemonic QBL for BRL.) JRS takes care of setting up the EXITLIST, and the various exits take care of removing the last exit from the EXITLIST and returning there.

Since a POP language exit is always taken from the bottom of the EXITLIST, a subroutine does not have to know who called it or where it is returning to. An interesting corollary of this is the following: if the last thing that one subroutine does (before exiting) is to call another subroutine, it is unnecessary to write

```
JRS  SUBR

BRU  EXIT
```

It is shorter and faster to write

```
BRU  SUBR
```

When SUBR exits, it will not return to the first subroutine, because no JRS was performed there. Instead, it will return directly to the caller of the first subroutine. This technique is widely used throughout the compiler. Care must be taken, however, if the first subroutine wants to return with an answer. Sometimes the second subroutine will return with the same answer, but more often not.

It is also possible for a machine language subroutine (e.g. called with a BRL) to recurse, by entering POP language via the instruction

```
BAL,IA    RECURSE
```

The words immediately following this instruction will be interpreted as POPs, and when an exit is performed, it will be to the POP immediately following the BRL that originally called the machine language subroutine. This is described further in Sections 1.9.2.2 and 1.9.3.2.

1.8.3 Tricky Recursive Coding

An example of a recursive subroutine that directly calls itself is the classic case of the factorial function. It can be coded straightforwardly as follows (assume the argument is in W0):

```
FACTORAL  SWG      =2
          BRL,F        EXITML
          FET      W0
          SUW      =1
          JRS      FACTORAL
          MPS      W1
          BRL      EXITML
```

One word can be saved (at a slight increase in time) by doing the following:

```
FACTORAL  SWG      =2
          FET,T        W0
          SUW,T        =1
          JRS,T        FACTORAL
          MPS,T        W1
          BRL      EXITML
```

Note that in this case, each level is remembering the answer flag it set up, even though it is recursing and setting up new answers with the same instructions.

A strange example of remembering the answer flag can be found in GENLIT, in the section SCONS, which processes alphanumeric strings. Here, the compiler wants to remember an answer over a range of quite a few POPs. However, some of those POPs want to make another test, which would destroy the answer being saved. Since there is no direct instruction for saving and recalling the answer flag, the easiest way to do this is to JRS, not to a subroutine, but to the instruction following the JRS. The coding which follows, although it has no intention of making use of the exit on the EXITLIST, can make use of the additional answer flag. When it is finished, instead of exiting, it does an MMO EXITBOT, which simply removes the exit along with the answer flag, and thus restores the original answer flag. This scheme takes advantage of knowing how exits work. However, this kind of "knowing how things really work" has to be done with restraint; otherwise the parameterized and modular characteristics of the compier are lost.

It can also turn out, becuase of the additional answer flags and the ability to "fall into" a subroutine, rather than calling it and then exiting, that time and space can be saved by making a piece of code a subroutine, even though it is only called once. ALTEMPS (Allocate Temps), in PASS3, is such a case, though it is fairly involved to show that this is true.

1.9 Interpreter INTERPTR

Section 1.6 discussed the use of POPs. This section gives further details on the operation of the interpreter (INTERPTR), how POPs are defined, and how they interface with the interpreter itself and the rest of the compiler. The following definitions will be in use:

A POP is an interpretive instruction, i.e., a word specifying an operation to be performed by the interpreter. For example,

```
        STK     ALPHA
```

A POPD (POP Definition) is the piece of code (generally written directly in machine instructions) called upon by the interpreter to perform the operation specified by the corresponding POP. For example,

```
STK     EQU     $-POPBASE
        LW,R    *WORKBOT
        STW,R       0,IA
        BIR,IL  INTERP
```

Note that both the POP and the POPD are named "STK", so the distinction must be observed carefully.

1.9.1 How to Write and Use POPs

1.9.1.1 Assembly format

The POP name is written in the Command field. If execution of the POP is to be conditional, the POP name is followed by ",T" (execute only if answer flag TRUE) or ",F" (execute only if answer flag FALSE). The address is written in the Argument field, preceded by "*" if indirect addressing is desired, and followed by ",X1" if indexing is desired. Use of the Label and Comment fields is the same as with machine instructions.

1.9.1.2 Special addresses

If the address written in the Argument field is one of the symbols listed below, the actual effective address is determined as follows:

Symbol	Effective address
Wn	(WORKBOT)-n

Pn (PARAMBOT)-n

PnIND ((PARAMBOT)-n)

where (a) means "contents of a". This is handled by the WAD, PAD and PAI POPs, described in Section 1.6.2.

Not all values of n can automatically be provided for. Only the values shown below are currently implemented.

Special address symbols

W0	P0	P0IND
W1	P1	P1IND
W2	P2	P2IND
W3	P3	P3IND
W4		

When one of these special addresses is used, indexing and indirect addressing may not be specified.

1.9.1.3 Word format

A POP occupies a 32-bit word as follows:

	*	X	T	F	op	address
Bits	0	1	2	3		

where:

- * is the indirect address specifier
- X is the indexing specifier (always X1)
- T indicates "execute if TRUE"
- F indicates "execute if FALSE"
- op is the POPD location relative to POPBASE

1.9.1.4 Effective address calculation

The effective address of a POP is computed by these steps (IA is the register used to hold the instruction address):

1. The 16-bit direct address (bits 16-31 of the POP) is placed in IA 16-31, and IA 0-15 is cleared.

2. If indexing is specified, (X1) 15-31 is added to (IA) 15-31. Note that a 17-bit index value is added to a 16-bit direct address (to which a leading zero has been appended).

3. If indirect addressing is specified, (IA) 15-31 is used as an address to access a word containing the indirect address. Bits 15-31 of this word are copied to IA 15-31, while IA 0-14 remains cleared. Note that the indirect address is accessed by a 17-bit address. If indexing was not specified, the most significant of that address is zero, so that the word containing the indirect address would have to be in the lower 64K of memory.

Note also that (as on the 9300) indexing precedes indirect, and that (as on Sigma) multi-level indirect addressing is not provided.

Even though the direct address specified by the POP has only 16 bits, all other addresses involved (the index value, the location of the indirect address word, and the indirect address itself) are full 17-bit addresses. Some implications of this are:

A. The POP language portions of the compiler will not run in memory above 64K, due to the inability of POPs to access those cells directly and to branch to locations in that region. However, lists may extend into this area, since all data in lists is accessed indirectly through list parameters located in the lower 64K of memory.

B. POP's may not specify indirect addressing through other POPs, since the low-order bit of the other POPs op field will be included in the 17-bit indirect address.

C. Negative direct addresses will not work properly. It is common to write "-1,X1" meaning that the effective address should be (X1)-1. But if this address is written with a POP, the direct address (-1) consists of only 16 bits and has the value X'0FFFF'. The index value is added as 17 bits, with the result that the effective address is (X1)+X'0FFFF', which differs from the intended value by X'10000'.

1.9.2 How to Write POPDs

1.9.2.1 Defining the POPD name

As mentioned above, each POPD has the same name as its corresponding POP. The value associated with this name is a 12-bit absolute integer giving the

location of the POPD relative to the fixed location POPBASE (contained in INTERPTR). It is this integer that appears in the op field (bits 4-15) of the POP. Thus the name may be defined by a line of the form

```
name    EQU  $-POPBASE
```

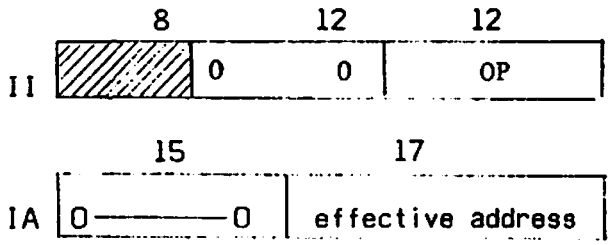
preceding the beginning of the POPD for "name". Since only 12 bits are available for representation of the op code, the POPD must begin in a loaction not more than 4K above POPBASE. This (and Meta-Symbol restrictions on the use of external REFs) necessitates that all POPDs begin in INTERPTR, wherein the symbol POPBASE is defined. If a POPD is to be written in some assembly other than INTERPTR, it should be defined within INTERPTR as a branch leading to the main body of the POPD outside INTERPTR. The symbol used for this purpose is, by convention, the POP name followed by the characters "POP". For example, if you want to write a POPD called BUZ, to be contained in PASS2, the INTERPTR should include:

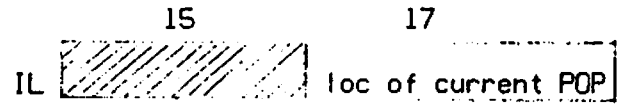
```
BUZ  EQU  $-POPBASE
     B    BUZPOP
     .
     .
     .
     DEF  BUZ
     REF  BUZPOP
```

The POPD itself, in PASS2, would begin at location BUZPOP.

1.9.2.2 POPD/Interpreter interface

Once the interpreter has fetched the POP word, determined that it is to be executed (in the case of conditional POPs), computed the effective address, and determined the op code, it branches to the POPD location with the registers set up as follows:





The POPD may use registers as shown below:

II	May be used as temp register.
IA	May be used as temp register, once the effective address has been used for its intended purpose.
IL	contains the interpretive location and should not be altered by the POPD (unless it wants to branch, of course).
AM	contains an address mask and should not be altered
A	is the POPs' A register and should only be referenced by POPs that are intended to make use of it (e.g. LDA, SERK).
X1	is the POPs' index register and should only be referenced by POPs that are intended to make use of it (e.g. LX1, POX).
X1,X3	Typically used as index temps.
R0-R6	Typically used as non-index temps.

Typical utility subroutines used by POPDs communicate arguments and results in R, and use X2, X3, and R1 as temps. Subroutines requiring a list number expect it in IA, since the POPDs that call those subroutines usually correspond to POPs whose effective addresses are list numbers (e.g. MON, CPY, RSV). The actual hardware registers used are shown in Section 1.9.2.3.

If the POPD is written in machine language (which it usually is), it returns to the interpreter via one of the following instructions:

```
No answer returned:        BIR,IL    INTERP
                                or
                           B         BIRINTRP

Set answer flag TRUE:      B         POPEXTRU

Set answer flag FALSE:     B         POPEXFAL
```

A POPD may be written wholly or partially in POP language. The POP language part is entered recursively, just as if it had been called with a JRS. It also exits just like a JRS-called routine, and this returns control to the next POP after the one whose POPD led to the recursive POP code. Such a POPD looks like this:

```
name EQU      S-POPBASE
      .
      . (machine language code)
      .
      BAL,IA   RECURSE
1A1   .
      . (POP language code)
      .
1A2   BRU      exitpl
```

Whre "name" is the name of the POPD and "exitpl" is one of the POP language exits described in Sections 1.9.3.2-3. The call on RECURSE causes the current value of IL (interpretive location) to be appended to the EXITLIST, and the location following the call is put into IL, causing POP interpretation to begin at location 1A1. When the exit at location 1A2 is performed, the EXITLIST bottom item is pulled off and placed in IL, thus returning control to the main sequence.

1.9.2.3 Register conventions in the Interpreter

Register	Name	Use
0	A	"A" register. Specifically referenced by certain POPs.
1	X1	POP index register. Set up by certain POPs.
2	X2	General index register.
3	X3	General index register.
4	II	Interpretive instruction code.
5	IA	Interpretive instruction address.
6	IL	Interpretive instruction location.
7	AM	Address mask (17 bits)
8	R (or R0)	General register
9	R1	General register

10	R2	General register
11	R3	General register
12	R4	General register
13	R5	General register
14	R6	General register
15	RL (or R7)	Subroutine link register. Also general register.

Although these register values are parameterized, and could conceivably be changed, certain properties are guaranteed, namely:

A. These pairs must be even-odd pairs: (X2,X3), (II,IA), (IL,AM), (R0,R1), (R2,R3), (R4,R5), (R6,R7).

B. X1, X2, X3, II, IA, and IL must be index registers.

1.9.2.4 Subroutines and special exits

Some utility subroutines of general interest to POPD writers are listed below.

Store on list bottom

Appends (R) to the bottom of list (IA).

Call: BAL,RL STRONBOT

Temps: X3

Does not alter R or IA.

Process pointer

Returns in X2 the absolute address of the item pointed to by the pointer in X2.

Call: BAL,RL PROCPNTR

Temps: X3

Fetch Flag word

Returns in R the flag word of the list (or plex or symbol) whose pointer is in R.

Call: BAL,RL FETFGWRD

Temps: X2, X3

Update Symbol Table pointer

The ID and mode fields (bits 0-11) of the ID word of the SYMTABLE item pointed to by the pointer in R replace the ID and mode fields previously in the pointer, leaving the displcaement unaltered.

Call: BAL,RL UPSYMPTR

Temps: X3, R1

Release list

Releases bottom file of list (IA).

Call: BAL,RL RLSLIST

Temps: X3, R

Does not alter IA.

Copy list

Copies bottom file of list (IA) onto list (PO), without removing the origin file. Discards PO.

Call: BAL,RL COPYLIST

Temps: X2, X3, R, R1

Does not alter IA.

Plain copy

Copies bottom file of list (IA) onto list (X3), leaving origin file alone.

Call: BAL,RL PLAINCPY

Temps: X2, R, R1

Does not alter IA or X3.

POPDs may also exit to the following locations, as well as to the standard exit points INTERP, POPEXTRU, and POPEXFAL.

SOBWEX	Store (R) on WORKLIST bottom and exit.
SOBWEXTR	Store (R) on WORKLIST bottom and exit TRUE.
SOBANDEX	Store (R) on bottom of list (IA) and exit.
SOBCOEX	Store (R) on CODELIST bottom and exit.
EMPLSEXF	Empty list (IA) and exit FALSE.
RELEASEX	Release list (IA) and exit.
RELSTEXF	Release list (IA) and exit FALSE.

1.9.2.5 Summary of POPD definition procedures

In the following, let XYZ be the POPD name.

To define a POPD within INTERPTR:

1. Add to the POP Proc in SYSTEM FORTRAN:

```
            REF   XYZ
XYZ         CNAME
```

2. Add to INTERPTR:

```
            DEF   XYZ
XYZ         EQU   S-POPBASE
            .
            . (POPD code)
            .
```

To define a POPD external to INTERPTR:

1. Add to the POP Proc in SYSTEM FORTRAN:

```
            REF   XYZ
XYZ         CNAME
```

2. Add to INTERPTR:

```
            DEF   XYZ
            REF   XYZPOP
XYZ         EQU   S-POPBASE
            B     XYZPOP
```

3. Add to the assembly that is to contain the POPD:

```
            DEF   XYZPOP
XYZPOP      .
```

1.9.3 How to write interpretive code

1.9.3.1 Machine/POP language interface

The initial entry into POP code is made as follows:

```
.
. (machine language code)
.
BAL,IL    ENTER
.
. (POP language code)
.
```

The BAL to ENTER causes the next instruction to be executed interpretively as a POP. Once started in this manner, POP execution will continue until a BRL POP (Branch and leave interpretive mode) is reached. BRL causes the interpreter to branch to the effective address, thus leaving POP language and beginning to execute machine code.

1.9.3.2 Subroutine linkage

POP language subroutines are entered recursively by the JRS POP (Jump Recursively to Subroutine). JRS causes the current POP location (contents of IL) to be appended to the bottom of the EXITLIST, and then branches (interpretively) to the effective address. The subroutine generally exits by performing a BRU to one of the POP language exits listed in Section 1.9.3.3, or a BRL to the corresponding machine language exit. (The machine language exits are equivalent, but are preferred because they are slightly faster.) At this point, the appropriate terminal function (if any, see below) is performed, the call address is removed from the bottom of the EXITLIST, and interpretive control is resumed at the call address plus one. There are a number of special exits available that perform various terminal functions; these are shown in the following section.

Alternatively, it is common for one subroutine to exit through another POP language subroutine which shares a common end-portion. The first subroutine simply BRUs to the second, which will exit properly since an exit is an exit; it does not matter who does it. A POP language subroutine may exit through machine code by performing a BRL to the machine code, which then returns to the interpreter by branching to one of the machine language exits shown in the following section.

The following table summarizes the various linkage techniques. These symbols are in use:

ML	Machine language
PL	POP language
subr	Subroutine being called
exitpl	Any of the POP language exits shown in the following section
exitml	Any of the corresponding machine language exits

1.9.3.3 Recursive Exits

Recursive subroutines often need to return an answer. Therefore, special exits are provided for sending back either TRUE or FALSE as the answer to the calling level. Sometimes, the answer depends on a test made near the end of the subroutine. Rather than having to write

```
QZR     something
BRL,T   EXITTRML
BRL     EXITFAML
```

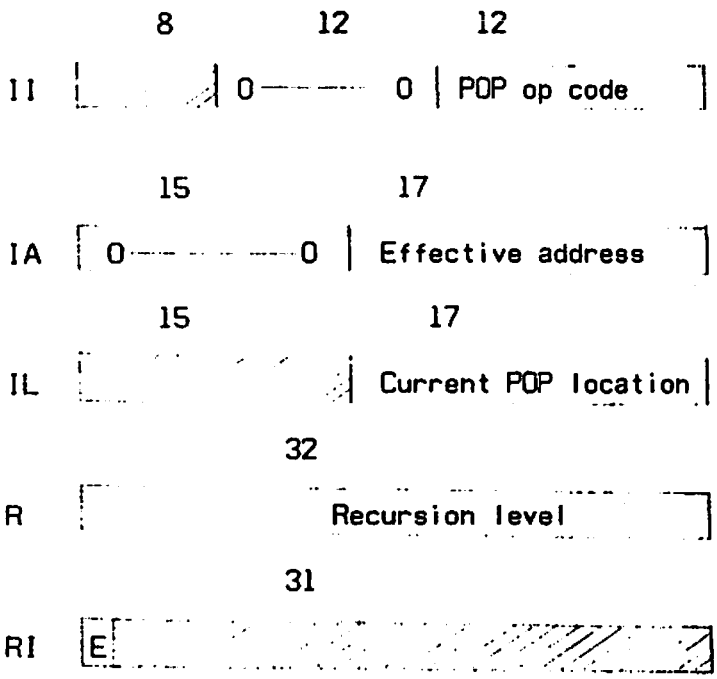
there are also exits provided that return the same answer as the last test (i.e. they pass the current answer flag up to the next level). It is also common that subroutines are left with superfluous values on either the WORKLIST or PARAMLST that need to be removed before exiting. A variety of special exits are provided for this, in combination with returning various kinds of answers. These are listed in the following table.

* 'CLEAR' removes values from WORKLIST. 'DITCH' removes values from PARAMLST.

PL exit	Corresponding ML exit	Answer	WORKLIST	PARAMLST
EXIT	EXITML	none	0	0
EXITTRUE	EXITTRML	TRUE	0	0
EXITFALS	EXITFAML	FALSE	0	0
EXITSA	EXITSAML	same	0	0
CLEAR1EX	CLR1EXML	none	-1	0
CLEAR2EX	CLR2EXML	none	-2	0
CLEAR3EX	CLR3EXML	none	-3	0
----	CLR4EXML	none	-4	0
CLR1EXTR	CLR1ETML	TRUE	-1	0
CLR2EXTR	CLR2ETML	TRUE	-2	0
CLR3EXTR	CLR3ETML	TRUE	-3	0
CLR1EXFA	CLR1EFML	FALSE	-1	0
CLR2EXFA	CLR2EFML	FALSE	-2	0
CLR3EXFA	CLR3EFML	FALSE	-3	0
CLR1EXSA	CLR1ESML	same	-1	0
CLR2EXSA	CLR2ESML	same	-2	0
CLR3EXSA	CLR3ESML	same	-3	0
DITCH1EX	DCH1EXML	none	0	-1
DITCH2EX	DCH2EXML	none	0	-2
DITCH3EX	DCH3EXML	none	0	-3
DCH1EXTR	DCH1ETML	TRUE	0	-1
DCH2EXTR	DCH2ETML	TRUE	0	-2
DCH3EXTR	DCH3ETML	TRUE	0	-3
DCH1EXFA	DCH1EFML	FALSE	0	-1
DCH2EXFA	DCH2EFML	FALSE	0	-2
DCH3EXFA	DCH3EFML	FALSE	0	-3
DCH1EXSA	DCH1ESML	same	0	-1
DCH2EXSA	DCH1ESML	same	0	-2
DCH3EXSA	DCH3ESML	same	0	-3

1.9.4 Interpreter/Debug Interface

If INTERPTR is assembled with DEBUG=1, it is prepared to interact with the compiler debug package, F4DEB. It does so only if the user requested the CDEB option on his control card (indicated by the DEBUGTRG being TRUE). In this case, after the interpreter has fetched each POP, determined whether or not it is to be executed, and computed its effective address, it then calls F4DEB, passing to it the following information in the registers shown:



where "Recursion level" is the depth of recursive nesting, indicated by the size of the EXITLIST, and "E" is the execution trigger (1 means POP is to be executed, 0 means it is not).

The contents of II, IA, and IL should not be altered by F4DEB unless the Debug user specifically intends to modify the action of the POP about to be executed.

1.10 Plexes

Plexes are a kind of structured data used to represent many things, especially arithemtic expressions and logical expressions. A plex consists of an operator and any number of operands. There are two parts to a plex: its pointer (which is usually on the bottom of the WORKLIST) and its data (which is found on the PLEXLIST).

The pointer tells some of the things we want to know about the plex. In particular, the pointer tells what the operator is. For example, the pointer may be a multiplication (*) or division (/) operator, and this fact is carried

in the pointer itself, in the portion that normally contains the list number.* This information is also repeated in the data on the PLEXLIST. So we can find out what kind of plex we are dealing with by using the SOT POP. Plexes also contain traits (sometimes, but not usually, called flags), and these can be tested with the SOF POP. The traits are part of the data contained on the PLEXLIST, and the SOP POP automatically knows where to get the trait information since the pointer in WO has an ID field which lies in the range for plexes.

The displacement portion of the plex pointer contains the position of the first word of data for this plex on the PLEXLIST. This position is relative to the Base of the PLEXLIST. The other field of the plex pointer is the mode field (i.e., INTEGER, REAL, or whatever) and this can be tested using the SOM POP. This mode information, like the ID field

---

* In the spectrum of ID numbers, there exists a range reserved for plex pointers. There are twenty or thirty such numbers, and they all implicitly point to the PLEXLIST. We never create a pointer which has the number of the PLEXLIST in its ID field.

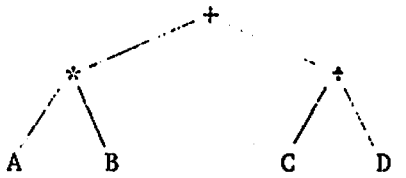
information, is also replicated in the data on the PLEXLIST.

The information about a plex kept on the PLEXLIST begins with a word count. This is the total number of words needed for the plex, including the word count itself. Next follows a word containing the plex number (ID field), mode, and twenty traits. Then follow the operands of the plex, which are usually pointers themselves. Sometimes these are pointers to lists, and sometimes they are pointers to other plexes.

1.10.1 Representation of Expressions

Plexes are naturally recursive in nature. That is, the operands of one plex can be pointers to other plexes. This makes a natural way to represent arithmetic and logical expressions, which are defined recursively.

The basic unit of representation is the plex. A plex is a collection consisting of an operator and all of its operands. For example, the expression A*B+C/D is represented by the following tree structure:



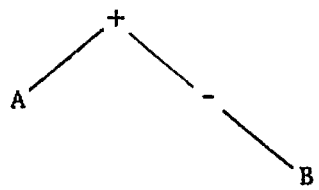
The expression consists of a Sum Plex whose two operands are a Product Plex and a Quotient Plex. The operands of the Product Plex are the two scalars A and B, while those of the Quotient Plex are the scalars C and D.

When Expression Scan scans the above expression, it leaves a pointer to the Sum Plex on the bottom of the Worklist. The rest of the information about the expression is contained on the PLEXLIST. The information on the PLEXLIST is as follows:

Relative Location	Contents	Remarks
0001	00000004	4 words in the plex (2 operands)
0002	5B200000	Product (5B), Real (2), no traits
0003	E1200001	Scalar (E1), Real (2), location 1 in SYMTABLE; A
0004	E1200007	Scalar; this one is B
0005	00000004	4 words in this plex (2 operands)
0006	5C200000	Quotient (5C), Real (2), no traits
0007	E120000D	Scalar; this one is C
0008	E1200013	Scalar; this one is D
0009	00000004	4 words in this plex (2 operands)
000A	5A200000	Sum (5A), Real (2), no traits
000B	5B200001	Product Plex, location 1 in PLEXLIST
000C	5C200005	Quotient Plex, location 5 in PLEXLIST

The expression A+B+C+D is represented as one Sum Plex containing four operands. (There would be six data words on the PLEXLIST) The code generator can use the four operands in any order it chooses. However, if the expression were written (+B) + (C+D), it would be represented as one Sum Plex combining two other Sum Plexes. This informs the code generator that it may not combine A with C or B with D, which it would have been able to do without the parentheses; nor may it add A to B, then add C to the result, then add D to the result; it must combine C and D with each other before it combines them with anything else. In this way, the integrity of the user's parenthesis groupings is preserved.

The expression A-B involves two plexes. It is treated as if it were written A + (-B) and consists of a Sum Plex combining the scalar A with a Minus Plex that has a single operand: the scalar B. Thus its tree structure would look like this:



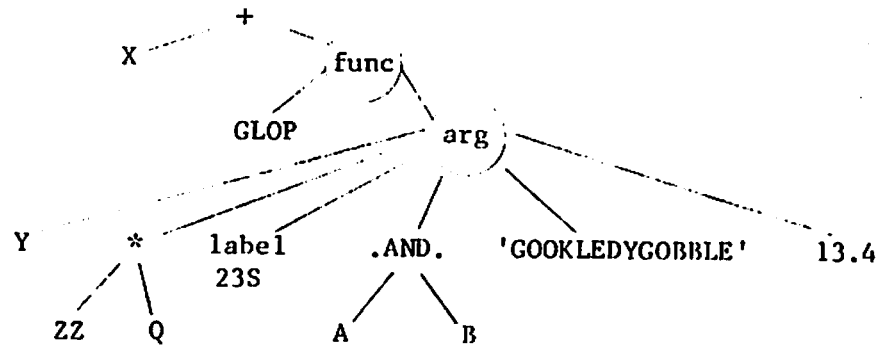
---

* The value on the WORKLIST points here.

In summary, there are five kinds of plexes involved in a basic arithemtic expression: the Sum Plex and Product Plex, containing a variable number of operands, but always at least two; the Minus Plex involving one operand; and the Quotient Plex (/) and Expon Plex (**) involving two operands.

In the previous examples, the only operands have been scalar variables, but there are other kinds of operands than scalar variables. Constants are represented as pointers to the appropriate constant list (there are six: the CKCONLST, HCONSLST, ICONSLST, RDCONLST, SCONSLST, and XCONSLST). Two other operands are array elements and function calls.

A function call is represented with a Function Call Plex (FUNCPLEX) which contains two operands: a Symbol Table pointer to the function name and a pointer to a Subprogram Argument Group Plex (ARGPLEX). The latter is a variable-sized plex and may contain as few as 0 arguments or as many as desired. Each argument is an expression pointer of some kind: a subscripted array plex, an array name pointer (unsubscripted), a statement label pointer, a subprogram name pointer, or a string constant pointer. Thus the expressionn X+GLOP (Y,ZZ*Q,23S,A.AND.B,'GOOKLEDYGOBBLE',13.4) comes out with the following tree:



There are two kinds of plexes involved in array elements: the Constant Subscripted Array Plex (CONSUBPX) and the Fully Subscripted Array Plex (FULSUBPX). A Constant Subscripted Array Plex is formed whenever the effective subscript (produced from combining all the subscript expressions) is a constant. This means that all the subscript expressions must be constant (for subscripting purposes, 3+5-2 is considered a constant), and the dimensions must be constant. Constant subscripts are not enough if the dimensions are adjustable. The Constant Subscripted Array Plex consists of two operands: a symbol table pointer to the array name, and a pointer to the Integer Constant List (ICONSLST).

A Fully Subscripted Array Plex (FULSUBPX) contains three operands: a symbol table pointer to the array name, a pointer to the Integer Constant List (ICONSLST), and a pointer to the non-constant part of the subscript. This will be an expression of INTEGER mode with most of its constants weeded out.

Incidentally, both the CONSUBPX and FULSUBPX can have a Multiple Dummy name instead of an Array name.

The plexes involved in logical expressions are quite analogous to the others. The .AND. Plex, .OR. Plex, and .EOR. Plex are variable-sized like sum and

product. The .NOT. Plex is unary like minus, and the .NE., .LE., .GT., .EQ., .GE., and .LT. Plexes are binary. The Extended Relational Plex (for constructs like A.LT.B.LT.C) is variable sized, but always contains an odd number of operands. The first, third, fifth, etc. operands are true operands, while the second, fourth, etc. are operators. The operators are the addresses (in LISTDEFS) of the plex constants that would have been used if the relational were not extended.

1.10.1 Plexes used to represent arithmetic expressions

The following is a table of the kinds of plexes used in representing arithemtic expressions. A separate table will be given later for plexes involved in logical expressions. A few plexes, such as Function Call Plex and Replacement Plex are involved in both kinds of expressions.

Table 1.10.1 Arithmetic Plexes

Short Name	Long Name	Meaning	Number of Operands
MINUSPLX	Minus Plex	Negation	1
SUMPLEX	Sum Plex	Addition	2 or more
ADDPLEX	Add Plex	Addition. Has same meaning and same plex number as SUMPLEX, but always has exactly two operands.	2
PRODPLEX	Product Plex	Multiplication	2 or more
MULPLEX	Multiply Plex	Multiplication. Has same meaning and same plex number as PRODPLEX two operands.	2
DIVPLEX	Quotient Plex	Division	
EXPONPLX	Exponent Plex	Exponentiation (**, raising to a power)	2
REPLPLEX	Replacement Plex	Replacement into left variable(s) of expression on right.	2 or more
SREPPLEX	Simple Replacement PLEX	Same as REPLPLEX, but two operands: one variable and one expression.	2
FUNCPLEX	Function Call Plex	Function Calling. The first operand is a Symbol Table pointer to the name of the function. The second is a pointer to a Sprog Arg Group Plex (ARGPLEX). The Function Call Plex is also used in the CALL statement.	2
ARGPLEX	Subprogram Argument Group Plex	Fastens together the arguments for a call on a subroutine or a function. Used as 2nd operand for a FUNCPLEX.	0 or more

(Blank Page)

1.10.1.2 Plexes used to represent logical expressions

The following are the kinds of plexes used in representing logical expressions. All of those below can be generated by the expression scanner in Pass 1.

Table 1.10.2 Logical Plexes

Short Name	Long Name	Meaning	Number of Operands
NOTPLEX	.NOT. Plex	Logical negation	1
ANDPLEX	.AND. Plex	.AND. operator	2 or more
ORPLEX	.OR. Plex	.OR. operaotr	2 or more
EORPLEX	.EOR. Plex	.EOR. operator (Exclusive .OR.)	2 or more
GTPLEX	.GT. Plex	.GT. relational operator (also)	2
LTPLEX	.LT. Plex	.LT. relational operator (also)	2
GEPLEX	.GE. Plex	.GE. relational operator	2
LEPLEX	.LE. Plex	.LE. relational operator	2
EQPLEX	.EQ. Plex	.EQ. relational operator	2
NEPLEX	.NE. Plex	.NE. relational operator	2

Simplify Logical Expression is a routine in Pass 2 that is executed before a logical expression is generated into object code. It recursively tears an expression apart plex by plex, then puts it back together again. It can generate some new plexes that were never seen or produced by Pass 1:

Table 1.10.3 Logical Plexes (Pass 2 Only)

Short Name	Long Name	Meaning	Number of Operands
EEORPLEX	Easy .EOR. Plex	An .EOR. plex all of whose operands are addressable	2 or more
HEORPLEX	Hard .EOR. Plex	An .EOR. Plex with at least one non-addressable term. This has the same plex number as EORPLEX.	2 or more
GTOPLEX	.GT. Zero Plex	Expression compared with zero	1
LTOPLEX	.LT. Zero Plex	Expression compared with zero	1
GEOPLEX	.GE. Zero Plex	Expression compared with zero.	1
LEOPLEX	.LE. Zero Plex	Expression compared with zero	1
EQOPLEX	.EQ. Zero Plex	Expression compared with zero.	1
NEOPLEX	.NE. Zero Plex	Expression compared with zero	1

1.10.2 POPs Concerned with Plexes

There are two sets of POPs associated with plexes: the plex-building POPs and the plex-destroying POPs. Of the plex-destroying kind there is the PUL POP. This POP expects a plex pointer in WO and removes it. It then "pulls" to the list e addressed by the POP the operands of the plex, however many of them there may be. When you know how many operands the plex has (e.g., the division plex always has two), it is a safe and good practice to pull to the WORKLIST. The pointer in WO is removed before the pulling occurs, so to pull a two-item plex to the WORKLIST is to increase the size of the WORKLIST by one.

When the number of operands is unknown (e.g., the Sum Plex may have any number of operands from two on up), it is customary to reserve a list and then PULl to it. In fact, the sequence

```
RSV     SOMELIST

PUL     SOMELIST
```

was so common in the 9300 compiler that on the Sigma we created a new POP called RPL (Reserve and PUll) which is used frequently in Pass 2. Thus

```
RPL     SOMELIST
```

is equivalent to the above sequence of POPs. This will not work where SOMELIST is WORKLIST.

On the plex-building side, there are a number of POPs. These POPs fall into two calsses: those that build with a fixed number of operands and those that build with a variable number.

The fixed-number-of-argument plex-building POPs are called FIP and FIC and work off the WORKLIST. They take the bottom n words from the WORKLIST, remove them from the WORKLIST, build a plex out of them, and append the newly created plex pointer to the bottom of the WORKLIST. Thus the POP

```
FIP     DIVPLEX
```

is rather the inverse of the pulling to the WORKLIST we discussed three paragraphs ago. There are two operands on the bottom of the WORKLIST (the numerator and denominator, actually), and the FIP POP removes these, builds a plex out of them, and puts the new pointer on the bottom of the WORKLIST. So

```
FIP     DIVPLEX
```

decreases the size of the WORKLIST by one word.

The FIP POP knows that there are two operands to the DIVPLEX: this fact is contained in the two-word data structure called DIVPLEX. It is a pair of constants (contained in the assembly called LISTDEFS) that direct the FIP POP

how to build the plex it has to build. These words contain the ID field to go into the plex. They have several possibilities for the mode field. They may have their own mode specific to the plex. (For example, the FIXPLEX always has mode INTEGER.) Or they may "inherit" mode from the mode of the operand(s). Then there are also traits, some of which may be built into the plex and others of which may be inherited. For both the mode and trait fields, the first word tells what is inherent in the plex and the second word tells what can be inherited.

1.10.2.1 Plex-building POPs

There are five plex-building POPs. Two of them--CIC and CIF--guild variable-sized plexes, and the other two--FIP and FIC--build fixed sized ones. The fifth POP--called PLO--is a sort of auxiliary plex builder. The name means PLex Open, and the POP is used in conjunction with CIC and CIF.

Variable-sized plexes, such as SUMPLEX, PRODPLX, and ANDPLEX are built from lists which have typically been reserved, and the plexes are built from the bottom file of such lists. One writes

```
CIC       SUMLIST
```

to build a Sum Plex from the bottom file of the SUMLIST. With variable-sized plexes it is necessary to speicy what kind of plex to build, as well as where to build it from. To specify the former, one writes PLO, as in

```
PLO       SUMPLEX
CIC       SUMLIST
```

Having built the kind of plex they were told to, CIC and CIF append a pointer to it onto the bottom of the WORKLIST. The only difference between CIC and CIF is that CIC inherits traits from all the terms in the plex, whereas CIF inherits only from the first (top-most) term. CIC and CIF empty the bottom file but do not rleease the list.

FIC and FIP build their plexes from the bottom few items on the WORKLIST It is inherent in each fixed-sized plex how many terms it involves, from Minus Plex (MINUSPLX) with one term all the way up to Data DO Control Plex (DVLWDPX) with six, and that many terms are taken from the bottom of the WORKLIST and built into a plex. The pointer to the plex is appended to the bottom of the WORKLIST after the other terms have been removed. It takes only one POP to build a fixed-sized plex, e.g.,

```
FIP       DIVPLEX   Quotient Plex
```

The only difference between FIP and FIC is that FIP inherits traits from all the terms in the plex, while FIC inherits only from the first (top-most) term.

It can be seen that for building a one-word plex like Minus Plex, FIP and FIC will have the same effect.

It should be noted that the inheritance of traits is not important in some plexes. Certain plexes, such as the ones used in the DATA statement , are used only to indicate structure. Similarly, the Sprog Arg Group Plex (ARGPLEX), the Dummy Group Plex (DMYGRPX), and the REPEAT Triple Plex (TRIPLEX), only indicate structure. The main use of traits is in plexes used for expressions.

Another peculiarity exists in the plex-building POPs. When any of the POPs is about to append a new plex to the bottom of the PLEXLIST, it first searches the entire bottom file (probably the only file) of the PLEXLIST to see if there is another plex just like it. If there is such a plex (and it must agree in all words), we do not put a new plex on the PLEXLIST. Instead, we create a new pointer to the old plex.

This feature saves a few words now and then on the PLEXLIST BUT ITS RAISON D"ETRE IS NOT TO SAVE SPACE but to provide an easy way of recognizing common subexpressions. If two expressions are identical, the pointers to them will be identical. Equality of expressions is checked for in subscripts, and even horribly messy expressions can be easily recognized as identical.

1.10.2.2 Plex-destroying POPs

In order to use a plex that has been built it is necessary to destroy it--to break it down and look at the pieces. After one has found out what kind of plex it is (by SOTing the plex pointer in WO) and perhaps examined the traits and mode of the plex (with SOF and SOM), one probably wants to break the plex, and there are two POPs for doing this: PUL and RPL.

PUL is the basic one and stands for "pull." This means that it pulls the terms of the plex to some list. It expects a plex pointer in WO, and it removes that pointer before doing the pulling. Thus it is legitimate to PUL to the WORKLIST, provided you know how many terms you are pulling. If you do not know how many terms there are, you had better pull to some other list. It is likely that you will want to reserve this other list before pulling (since you are apt to be involved in a recursive situation). If so, you can save one POP by writing RPL-- Reserve and PuLl.

RPL will not do anything useful if it addresses the WORKLIST. The usual list to pull things to (in Pass 2, at least) is the TERMLIST.

1.10.3 Indexing by Plex Type

In Pass 2 a popular way of branching to one of several possible code generation routines (based on the kind of plex pointer we have) is to index by

plex type. Such indexing is based on the fact that plex numbers form a sequential set, and they were purposely arranged that way.

The main POP for doing plex indexing is QIX (Query ID field into Index). This POP (described in detail in 1.7.2.9) takes the ID field of the pointed in WO and loads it into Index 1, then checks to make sure it is in the proper range (i.e., that the pointer is of one of the plex types we are interested in). The following example, taken from Simplify Logical Expression, will serve to illustrate the technique of indexing by plex type:

```
SIMPRANG  EQU  RANGE(EXRELPXN,NEPLEXN)  Defines the
          .                             range of plex types we are
          .                             interested in
          .
          QIZ  SIMPRANG                 Get ID to X1; check range
          JRS,F     P2ABORT             Abort if not in range
          DXI  LOW(SIMPRANG             Lower index range to start at 0
          BRU  SIMPJUMP,X1              Do indexed branch
```

The JRS is there in case something goes wrong (due, probably, to a compiler bug or hardware failure) and the pointer is not what we expect it to be. The DXI is there because of our stated inability to correctly express certain negative addresses, as would be required to do the (seemingly equivalent) unbiased jump:

```
          BRU       SIMPJUMP-EXRELPXN,X1
```

(See "POPs", 1.6, and "Effective address calculation", 1.9.1.4). LOW(SIMPRANG) is equal to EXRELPXN, but better parameterized.

This kind of indexing is used not only for plex types, but also for the Pass 2/3 interface items, which also are identified by their ID field values. Both Assemble Code (ASMCODE) in Pass 2 and Output Program (OUTPROG) in Pass 3 do this kind of indexing based on Pass 2/3 interface item types.

1.10.4 The Plex definition Procs

In LISTDEFS, there are a number of two-word constants that define the plexes. They tell the plex-building POPs how to build the plexes. These constants are defined very parametrically. Two different Proc names are used to define these constants--PLEX and SAME. Of the two, PLEX is by far the more common.

Two separate PLEX lines are usually used to define each two-word constant. CF(2) indicates whether this is the first or second of the two lines. If there is no CF(2), then this is the first line. Being the first line, it automatically causes the plex number to be advanced. Plex numbers are somewhat different from trait values and mode value (see Section 1.13) in that in general they do not have separate names for the value and the constant. There is an assembly parameter called LN (List Number) that is continually getting advanced. It is advanced during the definition of the lists, and it also gets advanced during the definition of the plexes. By omitting CF(2), you are indicating that you want LN advanced by one and this value put into the first field (8 bits) of the generated word. This is the plex number that will be built into any plex pointers. Next comes the mode field, which is in AF(1). If this plex has a specific mode that is not derived from the constituents of the plex, then that mode value is put here. This is done parametrically, such as with INTG for INTEGER (see Section 1.13.2). If there is no mode (it is inherited from the terms of the plex), this parameter is zero.

Next come the traits that are indigenous to the plex--those that are there regardless of what other traits may be inherited. These are written in AF(2) through AF(n), however many there are. Maybe there are none.

Into CF(2) of the second PLEX line is put the number of arguments to the plex. This may be from 1 to 6 if this is a fixed-number-of-arguments plex. A higher value than 6 may be used, but 6 happens to be as high as we go. If a variable sized-plex is intended, such as would be built with CIC or CIF, the value 0 is put here.

Then in AF(1) comes the mode field. HERE THE FIELD IS SOMEWHAT DIFFERENT, BECAUSE AN ACTUAL MODE VALUE IS NOT used, but a mask: either all zeroes or all ones, written as 0 or -1. This indicates whether or not the mode is to be inherited.

As in the first word, AF(2) through AF(n) are traits, but these are traits that are not necessarily true of this plex but can be inherited from the terms of the plex. The first word indicates what is definitely in the plex, and the second indicates what may be inherited. This is true of both the mode and trait fields. For example:

```
SUMPLEX   PLEX 0

          PLEX,0     -1,ABNO,CPXM,DBLE,NEVX
```

The first line advances the plex number and puts it into the first byte, and zeroes in the rest of the word. There is no inherent mode and no traits that are indiginous to this plex. The second line, having the ",0" after the word PLEX, does not advance the plex number; it generates zero in the first byte, indicating an argument count of zero (variable-sized plex). The mode field is generated as X'F', indicating that the highest mode of the constituents is to be used. (The nice thing about using -1 for this value is that you do not have to know the size of the field.) The next four names are traits that can be inherited.

Sometimes we need to remember the number of the plex (which was generated as the first byte of the first word). We do this after the PLEX lines, e.g.,

```
SUMPLEXN EQU   LN
```

It happens that we want to generate another plex that has the number as SUMPLEX. This is called ADDPLEX and is just the same as SUMPLEX except that it has specifically two terms instead of a variable number. So we write

```
ADDPLEX SAME   0


        SAME,2 -1,ABNO,CPXM,DBLE,NEVX
```

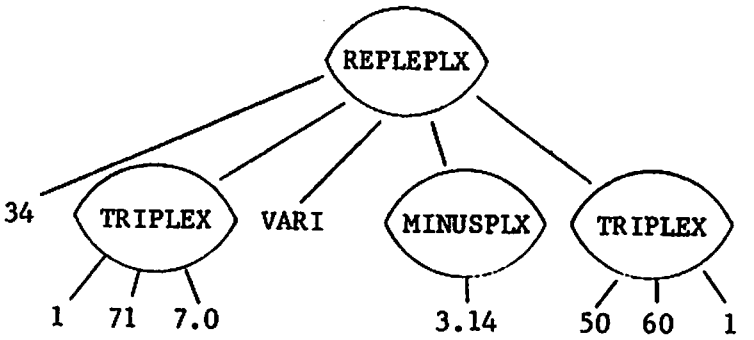
Again this is written <u>after</u> the PLEX lines we want it to be the same as. The SAME Proc is the same as the PLEX Proc except that it never advances LN.

1.10.5 Other Uses

In addition to arithmetic and logical expressions, plexes are used for representing a variety of structurish things. For example, the REPEAT FOR statement contains a list of items, each of which may be an expression or a triple of expressions. The triple is enclosed in parentheses and contains the initial value, termination value, and step size for an iteration. The three items of the triple are glued together with a Repeat Triple Plex (TRIPLEX). The whole list of items is held together with a Repeat List Plex (REPLSPLX). Let the statement be

```
REPEAT 3, FOR X = 34, (1,71,7.0), VARI,-3.14, (50,60,1)
```

Then the plex structure for the part of the statement after the "=" sign is



The list of labels in a computed GO TO statement are combined into a Label Group Plex (LBLGRPLX). Thus the above two cases where the Pass 1/2 interface uses plexes because they are a convenient way of treating a variable-sized list as if it were a single word. This is how we keep the number of words of interface data constant for any particular kind of statement.

There are a few other non-expression uses of plexes, but the most important are in the DATA statement. Here the recursive nature of plexes is required for both the variable lists and the constant lists. The variable part (left-hand side) can have implied DO-loops and parenthetical groupings nested to any depth, whereas the constant part (right-hand side) can have implied repetitions and parenthetical groupings to any depth.

During the scanning of a DATA statement, quite a few plexes get built. Then during Expand DATA Pair Sequence (EXPDPRSQ) the plexes get pulled apart and broken down. The scanning part is covered in Section 3.34.1, while the expanding is described in 3.34.2.

The tree structures produced for DATA statements can get quite complicated, as the following diagrams will show. Consider the statements

```
      REAL CROSS(10,20),ANGRY(5,6,7),X, Y

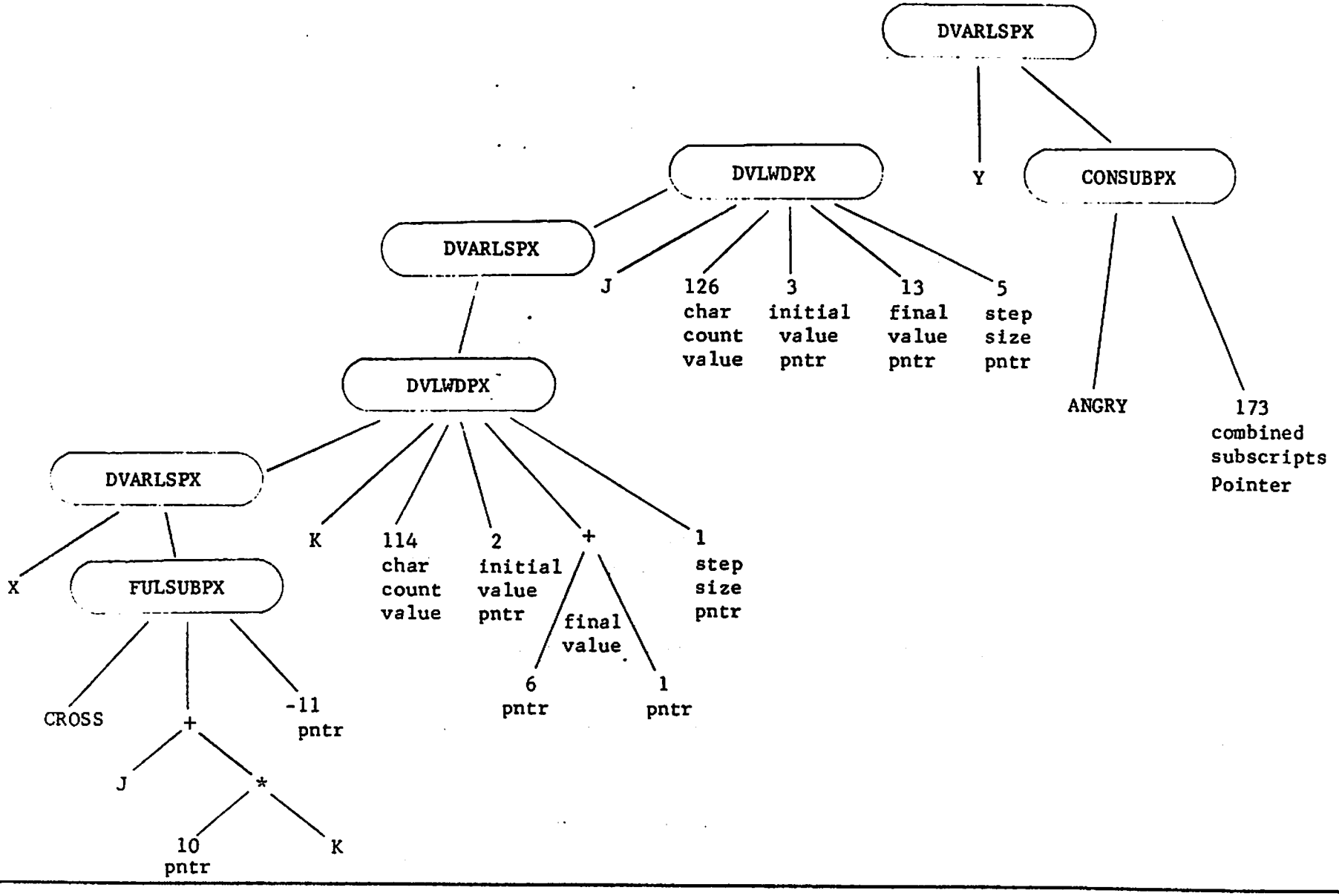
      DATA ((X,CROSS(J,K), K = 2,6+1), J = 3,13,5), Y, ANGRY(4,5,6)

     2     /2.718, 3*(24,-62.4), 2*(386, 7*48.33), 100/
```

The DATA statement is unrealistic considering that it initializes the variable X with eighteen different constants. However, the plex structures produced for the two halves of this DATA statement are shown in the following two diagrams. We must consider these to be the first three source cards of the program, in order for the character counts to come out correctly.

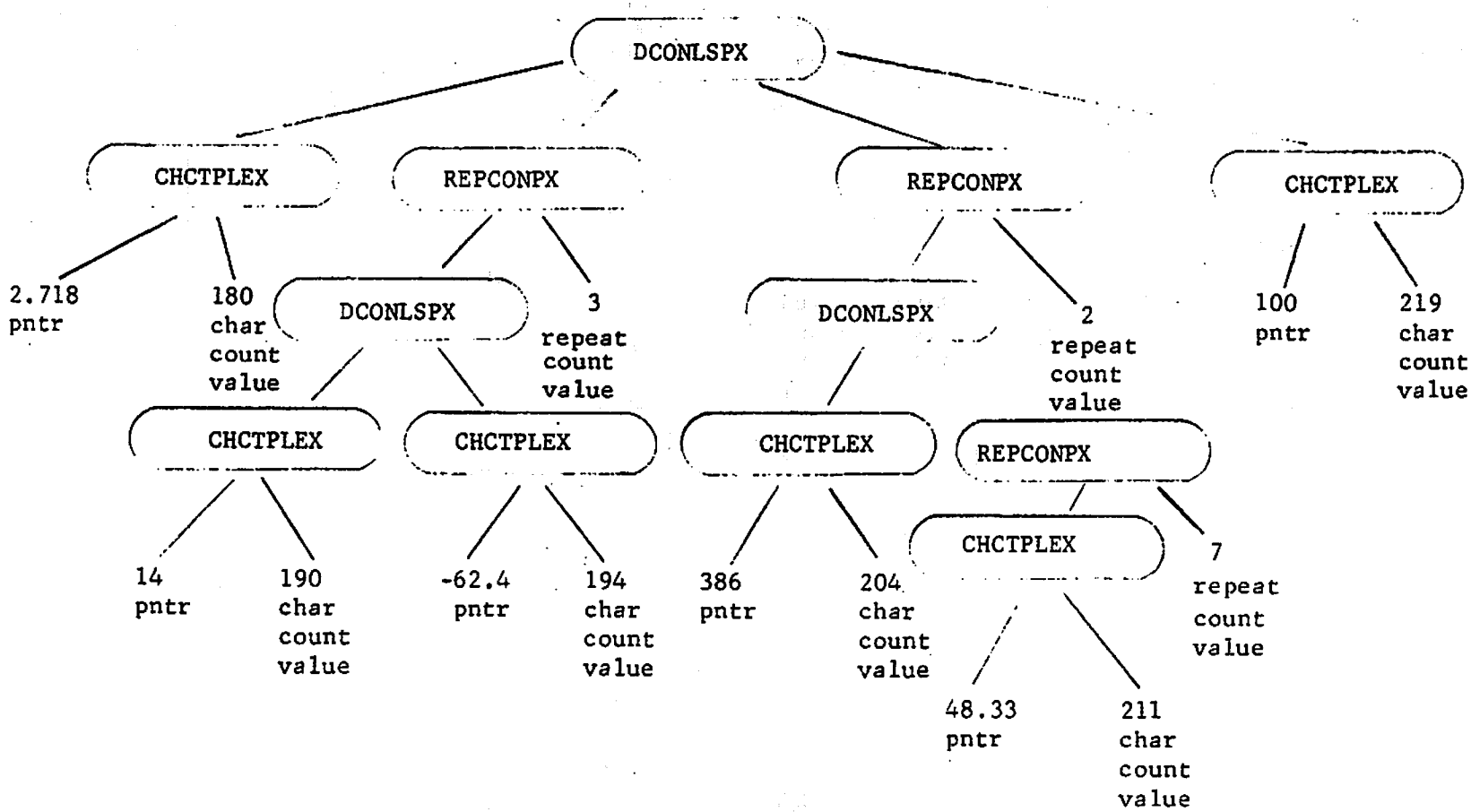
Plex structure for the variable portion of the following DATA statement:

```
 REAL CROSS(10,20), ANGRY(5,6,7), X, Y
 DATA ((X,CROSS(J,K), K = 2,6+1), J = 3,13,5), Y, ANGRY(4,5,6)
2    /2.718, 3*(14,-62.4),2*(386, 7*48.33), 100/
```



Plex structure for the constant portion of the following DATA statement:

```
      REAL CROSS(10,20),ANGRY(5,6,7), X, Y
      DATA ((X,CROSS(J,K), K = 2,6+1), J = 3,13,5), Y, ANGRY(4,5,6)
     2    /2.718, 3*(14,-62.4), 2*(386, 7*48.33), 100/
```



1.11 Pointers and Pseudo-Pointers

Much of the data that the compiler works with consists of pointers or things that look very much like pointers. A pointer looks like this:

ID Identification Mode Displacement

It has three fields--Identification, Mode, and Displacement. The Mode field, when used at all, means integer, real, etc.

A "classical" pointer points to a list. That is, it has a list number in its ID field and a dispacement (distance) from the Base of that list in its Displacement field. It points to something.

Some "semi-classical" pointers are Symbol Table pointers and plex pointers. Here the IDfield is not the number of the SYMTABLE or the PLEXLIST, but the displacements are still from the bases of these lists.

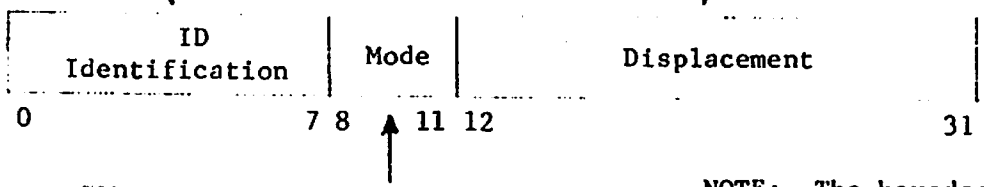
Then we get to some "pseudo-pointers" such as temp pointers. Here there is no list involved at all, and the displacement is from the start of a certain kind of temps at execution time. Nothing is really pointed "to".

Still less "classical" are the pointer-like things used in the Pass 2/3 interface data. Here the first two fields are all right, but the Displacement field may contain peculiar things such as the op code of a generated instruction.

The numbers that can appear as ID numbers form a spectrum from 00 to FF (not all of which are allowed), and from the ID field one can figure out what the Displacement field means. The following table shows the different classes and what is what.

Table 1.11.1 P O I N T E R S a n d S U C H

00	Zero List	Zero List Pointers--displacement = actual address within compiler
01-4D	Dynamic Lists	Static and Dynamic List Pointers--displacement from BASE of list
4E	not used	
4F-54	Static Lists	
55-57	Fake Lists with traits	Temp or Ptemp or Rttemp Pointers--displacement from start of temps or ptemps or rttemps
58	not used	
59-81	Plexes	Plex Pointers--displacement from PLEXBASE
82-94	Pass 2/3 Operators	Pass 2/3 Operators, Operands--various things
95-96	Pass 2/3 Operands	
97	'ALLOCATE'ID	EQSETBIT Word--EQUIVALENCE statement line number
98		Local Variable Pointer--displacement from start of local variables
99-DF	not used	
E0-E2	Symbol Table ID	Symbol Table Pointers--displacement from SYMTBBAS
E3	not used	
E4-E5	Symbol Table ID	
E6-ED	not used	
EE	Symbol Table ID	
EF-FF	not used	



0	Unknown or unimportant
1	Integer
2	Real
3	Double Precision
4	Comple
5	Double Complex
6	Logical
7	Label
8-F	not used

NOTE: The hexadecimal numbers given for ID fields correspond to the 12/18/69 version of LISTDEFS. In later assemblies these numbers may change, mainly through insertion of new entries into each category. The actual values are rather unimportant, since the whole thing is quite heavily parameterized.

1.12 The Spectrum of ID Field Numbers

The following table, occupying several pages, is a detailed expansion of the one just given. It shows just what each possible hexadecimal value for an ID field will mean. These actual hex values are not very important. It is more important to observe what order the different classes come in, and what order the plexes appear in, rather than to know actual hex values, which are subject to change with reassembly of LISTDEFS.

Two names are associated with each ID number. The names in the left-hand columns have the values shown. For example,

```
OUTLIST     EQU     01
```

The names in the right-hand column are of constants containing that hex value in the upper byte (and other information in the rest of the word). For example,

```
OUTCODE     GEN,8,24  OUTLIST
```

Thus a name in the right-hand column is suitable for SOTing against. The alternate names that sometimes appear to the right of the hex number are sometimes alternates to the left column, sometimes to the right. You just have to know.

The right-hand column is blank for many lists; the left-hand column is blank for most plexes. When a plex number symbol has a value, it is used for controlling jump tables in Pass 2 that are indexed by plex number. If both columns are blank, then the number is not used.

Table 1.12.1 The Spectrum of IO Field Numbers

Name of Value	Value in Hex	Constant containing Value	Alternate Names
L I S T S			
ZEROLIST	00	ZEROCODE	
D Y N A M I C L I S T S			
OUTLIST	01	OUTCODE	
INPUTLIST	02	INPUTCOD	
SYMTABLE	03	SYMTBCOD	
SAVELIST	04	--	
LOCLBLST	05	--	
DOLIST	06	--	
DSUBLIST	07	--	
CLUMPLST	08	CLUMPCOD	
EQUIVLST	09	--	
EQVCHLST	0A	EQVCHCOD	
ERRORLST	0B	--	
CKCONLST	0C	CKCONCOD	
CRELBLST	0D	CRELBCOD	
DECLLIST	0E	--	
DHEXLIST	0F	DHEXCODE	
GROUPLST	10	GROUPCOD	
HXONSLST	11	HCONSCOD	
ICONSLST	12	ICONSCOD	
LECOMLST	13	--	
LECOMLST	14	--	
LITERLST	15	--	
NLCLBLST	16	NLCLBCOD	
NAMELST	17	--	
RDCONLST	18	RECONCOD	
REFLIST	19	REFCODE	
SCONLST	1A	SCONSCOD	
SCRPTLST	1B	SCRPTCOD	
XCONSLST	1C	XCONSCOD	
ALPHALST	1D	--	
ANDLIST	1E	--	
ARGLIST	1F	--	

```
ARRAYLST  20                --
ASNLBLST  21                --
ASTLIST         22                ASTCODE
BLCOMLST  23                BLCOMCOF
BLNAMLST  24                BLNAMCOD
DPAIRLST  25                --
CODELIST  26                --
COMPGLST  27                --
CPXTMLST  28                --
DACONLST  29                --
DCODLIST  2A                --
DIMLIST         2B                --
DUMMYLST  2C                --
EASYLIST  2D                --
ENTRYLST  2E                --
EORCOLST  2F                --
EORLIST         30                --
EXITLIST  31                --
GLNAMLST  32                GLNAMCOD
DPRSZLST  33                --
DREPLIST  34                --
HARDLIST  35                --
IFLIST          36                --
KLUDGLST  37                --
MODELIST  38                --
INDOLIST  39                --
LABELLST  3A                --
ORLIST          3B                --
PARAMLST  3C                --
PLEXLIST  3D                --
PRODLIST  3E                --
PXEMBLST  3F                --
DVARLIST  40                --
REPLIST         41                --
REPLLIST  42                --
SCREXLST  43                --
SIGNLIST  44                --
SPLBLIST  45                --
SRCLNLST  46                --
SUMLIST         47                --
SUMRYLST  48                --
SVERRLST  49                --
SYTCHLST  4A                --
TERMLIST  4B                --
TRYLIST         4C                --
WORKLIST  4D                --                LASTLIST
*
*
*
*
*  S T A T I C   L I S T S
```

```
*
      OPCODLST  4F              --
      OPCSLIST  50              --
      INTRILST  51              --
      UNITLIST  52              --
      XREGLIST  53              XREGCODE
      DREGLIST  54              DREGCODE
*
*
*
*     F A K E   L I S T S   W I T H   T R A I T S
*
      --              55            TEMPNTR
      --              56            PTEMPPNT
      --              57            RTTEMPTR
      --              58            --
*
*
*
*     P L E X E S
*
      --              59            MINUSPLX
      SUMPLEXN  5A                  SUMPLEX       ADDPLEX
      --              5B            PRODPLEX      MULPLEX
      --              5C            DIVPLEX
      --              5D            EXPPLEX
      --              5E            MODESWPX      ISOLPLEX, FIXPLEX, INTGPLEX,
                      5E                                SNGLPLEX, DOUBLPLEX, CMPXPLEX,
                      5E                                KMPLEX
      EXBELPXN  5E                  EXRELPLX
      FUNCPLXN  60                  FUNCPLEX
      REPLPLXN  61                  REPLPLEX      SREPPLEX
      --              62            NOTPLEX
      --              63            ANDPLEX
      --              64            ORPLEX
      --              65            EORPLEX       EEORPLEX
      --              66            LTPLEX
      --              67            LEPLEX
      --              68            GEPLEX
      --              69            GTPLEX
      --              6A            EQPLEX
      NEPLEXN         6B            NEPLEX
      --              6C            HEORPLEX
      --              6D            LTOPLEX
      --              6E            LEOPLEX
      --              6F            GEOPLEX
      --              70            GTOPLEX
      --              71            EQOPLEX
      NEOPLEXN  72                  NEOPLEX
      --              73            CONSUBPX
```

```
--              74              FULSUBPX
--              75              COMPPLEX
--              76              TRIPLEX
--              77              ARGPLEX
--              78              LBLGRPLX
--              79              REPLSPLX
--              7A              DMYGRPX
--              7B              MDMYGRPX
--              7C              CHCTPLEX
--              7D              DVARLSPX
--              7E              DCONLSPX
--              7F              DVLWDPX
--              80              REPCONPX
HIPLEXNR  81            ADREPLEX
*
*
*
*   P A S S  2 / 3  O P E R A T O R S
*

P23HABS         82              (HABSINDC)      NAMES IN PARENTHESES ARE
P23OPCD         83              --              defined in 'Pass 2', not
P23IMMOP  84            (BIMWDS)   in 'Listdefs'.
P23BOPC         85              --
P23BOPC2  86            --
P23OPCS         87              --
P23DEF          88              (DEFINDIC)
P23ADND         89              (ADNINDC)
P23SOURC  8A            (SOURCIND)
P23END          8B              (ENDINDIC)
P23BLC          8C              (BLCINDIC)
P23FABS         8D              (FULLABSI)
P23NABS         8E              (NABSINDC)
P23DBGEN  8E            (DBGENIND)
P23CNL          90              (CNLINDIC)
P23PLEXL  91            (PLEXINDC)
P23DTORG  92            (DTORGIND)
P23DCON         93              (DATACONI)
P23PROG         94              (RPROGIND)
*
*
*
*   P A S S  2 / 3  O P E R A N D S
*

P23ABS          95              (ABSINDIC)
P23PRLOC  96            (PROLOCIN)

*
*
*
```

```
*   'ALLOCATE' ID FIELD VALUES
*
    EQSETID        97          EQSETBIT
    LOCVARID  98             LOCVARPT
    --             99          --          THRU DF

*
*
*
*   SYMBOL TABLE ID FIELD VALUES
*
    UNUSEDID  E0             UNUSED
    SCALARID  E1             SCALAR
    ARRAYID        E2          ARRAY
    --             E3          --
    MULOMYID  E4             MULTOUMY
    SPROGID        E5          SPROG
    --             E6          --
    --             E7          --
    --             E8          --
    --             E9          --
    --             EA          --
    --             EB          --
    --             EC          --
    --             ED          --
    ILLEGID        EE          ILLEGAL
    --             EF          --
```

1.13 Traits and Mode

List pointers and plex pointers all have traits and mode associated with them. The mode is built into the pointer itself (bits 8-11); the traits are not in the pointer and are found elsewhere. In the case of plex pointers, they are contained in the data of the plex (on the PLEXLIST). In the case of list pointers other than Symbol Table pointers, the traits are contained in a table called FLAGS which is indexed by list number. (The terms "flag" and "trait" are, regrettably, used interchangably in the compiler.)

Traits are one-bit items that are either TRUE or FALSE. Thers is a 20-bit field allocated for these in whatever word contains the traits. Modes are specific values in a 4-bit field. The values 0 thru 7 are recognized as valid. Sometimes mode values are used to index into a table, as with the BAM POP.

Modes are tested (compared for equality) with the SOM POP. Traits are tested (checked for being on) with the SOF (Set On Flag) POP. SOF works analogously to the BANZ machine instruction--it returns a TRUE answer if the AND of the things being compared is non-zero. All of the trait constants have just a single bit set except one: ALOCATED. This has five of them set, and thus SOF ALOCATED will be TRUE if any of the traits is present.

1.13.1 Traits

All the definition of traits is done in LISTDEFS. The trait values are given 4-letter names. For example,

```
DUMY EQU X'00002000'
```

These values are used parametrically around LISTDEFS in many places. Then there are the constants that have these values. For example,

```
DUMMY    DATA DUMY
```

Thus each trait has in effect two names. The two names for each trait and their descriptions are given in the following table.

Name of Trait Value	Name of Trait Constant	Description
ABNO	ABNORMAL	Abnormal. This trait name appears on many plexes, indicating that it can be inherited by these plexes. For all the attention that is paid to this trait, it is a pity that it does nothing at all and is in fact equated to zero. It was originally intended to indicate a subprogram with side effects or a variable that could change with interrupts--in other words, something that had to be referenced (called or picked up from memory) each time it was used. This fact will be useful for a kind of global optimization if ever installed.
ADDR	ADRSBLFG	Addressable. This trait indicates that something can be addressed by a machine instruction. It applies to scalars, array elements, constants, labels, and the like. This is as opposed to expressions, which in general are not addressable. See also SADR and TADR.
BCOM	BLANKCOM	Blank COMMON. An allocation trait, applies to scalars and arrays and indicates that they have appeared in a COMMON statement assigning them to blank COMMON.
CONS	CONSTFLG	Constant. This trait is true of constants, i.e., pointers to the CKCONLST, HCONSLST, ICONSLST, RDCONLST, SCONSLST, and XCONSLST.
CPXM	CPXMF	Complex Mixture. This is a peculiar trait that is true only of certain plexes. It is inherited from the constitutents in the plex not in the usual way (which is by ORing that trait for each of the constituents), but by comparing the modes of the constituents. For this trait to be true, there must be at least one complex (or komplex) term and one non-complex term in the plex. It is used in optimizing the code generated for sums, products, and quotients, even though there is special machinery for handling this trait built right into the plex-building POPs.
DBLE	DBLFLAG	Double in expression. Although this trait is not equated to zero, it might as well be. It is set up but never tested. It is a carry-over from the 9300 days in which there was no DOUBLE COMPLEX mode and it was necessary to

distinguish a complex expression that contained some DOUBLE PRECISION in it somewhere from one that did not. This trait is in the mode constant DOUBLE and hence is automatically registered when one REGs DOUBLE.

DTTA	DATA	Has appeared in DATA Statement. This trait, applicable to scalars and arrays, is important to ALLOCATE which is interested in checking for DATA in blank COMMON--not allowed.
DUMY	DUMMY	Dummy. This trait applies to scalars, arrays, and multiple dummies. IT CAN ALSO BE inherited by Constant Subscripted Array Plex (CONSUBPX) and FullySubscripted Array Plex (FULSUBPX).
EQVG	EQVGLOBL	EQUIVALENCEd to GLOBAL. This trait is set up in ALLOCATE for the convenience of SUMMARY, which must list these names somewhat differently from other names.
EVEN	EVENLOC	Even Location. This trait applies to variables that must lie on doubleword boundaries, i.e., scalars and arrays of DOUBLE PRECISION, COMPLEX and DOUBLE COMPLEX mode. It is important mainly in ALLOCATE.
EXRF	EXTREF	External Reference. Applies to subprograms that must be externally referenced.
EXTR	EXTERNAL	External. Indicates tha a subprogram name has appeared in an EXTERNAL statement. It is not clear from such usage whether the name is an external reference or definition.
GLOB	GLOBAL	Global. Indicates that the scalar has appeared in a GLOBAL statement and is a primary global, as opposed to a variable that is merely equivalenced to global (EQVG).
INTR	INTRNSIC	Intrinsic. Indicates that a particular function name is intrinsic and has been recognized as such. In this case a certain rung of the SYMTABLE is set to the number of of this intrinsic function, which is derived from its position in the INTRILST.
IOLU	IOLUSAFG	I/O List Unsubscripted. This trait applies only to arrays and indicates that the array name has appeared unsubscripted in an input/output list (indicating the whole array). This fact means that the number of elements must be known at execution time. In the case of dummy arrays with adjustable dimensions (the only case that matters).

this element count must be computed. This trait signals Pass 2 to generate code to compute the count at prologue time in the subroutine. Thus, if it is used unsubscripted more than once, the number of elements is calculated only once.

LCOM LABCOM Labeled COMMON. Indicates that a variable has appeared in a COMMON statement and is allocated in labeled COMMON.

LOCL LOCALVAR Local Variable. A trait set up by ALLOCATE indicating that a scalar or array has been allocated in local variable storage (as opposed to COMMON or GLOBAL).

NEVX NEVALXFG Not Evaluatable in Index. An expression trait affecting subscripts, indicating that they cannot be evaluated in an index register. This could be from some non-integer sub-expression, or a function call, or the use of the "**" operator (exponentiation), or several other things.

SADR SADRFLAG Signed Addressable. This trait applies to expressions and is similar in meaning to Addressable. It means that something is Addressable except for a possible minus sign. So minus an Addressable thing is Signed Addressable.

SDEF SPROGDEF Subprogram Definition. Applies to a subprogram name, indicating that the name is defined in this compilation either in a statement function, or in a SUBROUTINE, FUNCTION, or ENTRY statement. See also INDP below.

TADR TADRFLAG Test Addressable. This trait applies to expressions and means that a thing is Addressable for purposes of testing its value. Thus an Addressable thing in a relational with zero is Test Addressable, since condition codes can be set in one instruction.

INDP LOOPINDP Loop-Independent. A plex trait indicating that an expression is invariant during a single iteration of the current innermost loop. Thus, the expression can be regarded as "constant" for one particular iteration, and if

it is evaluated once during the loop it need not be evaluated again. This trait is derived during Pass 1 (in the assembly PASSOF in the part that massages subscript expressions). An expression is examined to see if it consists of integer constants and scalar variables, all of the scalars being the control variables of currently active DO-loops. (A control variable of a REPEAT loop is no good, because it is allowed to be changed during execution of the loop, whereas a DO-loop control variable is not.) If the expression consists of the things just mentioned, combined by the operators +, -, *, and /, then the plex for the epxression is given the LOOPINDP trait.

Simple scalars and constants are not given this trait; only plexes. This is the only case in the compiler where a plex acquires a trait after the plex has been built. The trait is just ORed in.

The numeric value of this trait is the same as that for SPROGDEF, described above. They were overlapped because we needed an extra bit for a trait and did not have it. No problem is caused because SPROGDEF applies only to subprogram names.

ALOCATED Allocated. This is the only composite trait, for it consists of five other traits. Its value has no name but is the OR of BCOM, EQVG, GLOB, LCOM and LOCL. It is used in ALLOCATE and means that a variable has been allocated. The five traits are the five means by which it might have been allocated.

1.13.2 Modes

Like traits, modes are defined (in LISTDEFS) by a two-symbol process. There is a 4-character symbol containing the value of the mode, such as

```
INT             INTG      EQU             1
```

Then there is the constant containing the mode value in the appropriate field, such as

```
                INTEGER   GEN,8,4,20      0,INTG,0
```

In fact, the "GEN,8,4,20" directive is not used, but instead a PROC named IMT is called. So in LISTDEFS one finds

```
                INTEGER   IMT             0,INTG,0
```

where IMT means "ID, Mode and Traits" and knows where the fields lie.

This may sound like an unnecessary amount of parameterization but on the 9300 the ID field was 6 bits instead of 8 the mode field was 3 bits, and the traits were 15. To convert to the Sigma, all we had to do was rewrite the IMT PROC (which is used frequently in LISTDEFS), and all our trait and mode words assembled correctly without change.

The eight different values for the mode field, together with their double names, are given in the following table.

Mode Value	Name of Mode Value	Name of Mode Constant	Description
0	UNKN	UNKNOWN	Unknown or unimportant. This is the initial mode of most identifiers, before we find out what mode they are. Some identifiers always have this mode: for example, the names of SUBROUTINEs; alos, names that appear in a subprogram's dummy list but are never used or given a type (mode). These dummies are generated in a special way so that they can accept arguments of any mode; since they are unused, they might as well accept anything. Certain plexes also have this mode, and it is the defualt mode for the CODE word for lists.
1	INTG	INTEGER	Integer.
2	SNGL	REAL	Real.
3	DOUB	DOUBLE	Double Precision or REAL*8.

4	CMPX	COMPLEX	Complex.
5	KMPX	KOMPLEX	Double Complex or COMPLEX*16
6	LOGL	LOGICAL	Logical.
7	LABL	LABEL	Label. Applies to statement labels and created labels, i.e., pointers to the IOCLBLST, NLCLBLST, and CRELBLST.

1.13.3 Inheritance of Traits

Inheritance--of both traits and mode--is a means by which the top levels of a tree structure find out what is going on, in a gross and non-structural way, at the lower levels--at the roots, so to speak. Through inheritance of the NEVX trait, for example, one can test at the top level of an expression whether that expression can be evaluated in an index register. If there is some part of the expression that cannot be evaluated in an index register, that fact is inherited up the tree structure where it can be tested at the top with a SOF POP.

In a sense, traits can be inherited from two sources: from plexes and from the constituents of plexes. For example, a Function Call Plex has the Not-Evaluatable-In-Index- register trait built into it, regardless of the arguments of the function. But the Sum Plex can inherit this trait from its constituents. If any of the terms of the sum plex have the trait, the sum plex has the trait.

Trait inheritance is an automatic feature of the plex-building POPs. There is a 20-bit field in each plex on the PLEXLIST for the traits. Each plex-defining constant (those things addressed by the plex-building POPs) consists of two words, each containing a 20-bit trait field. The first word tells which traits are inherent in the plex. The second tells which traits may be inherited from the terms of the plex.

The second word may be thought of as the selector word. 1-bits indicate that the traits come from inheritance; 0-bits indicate that they come from the plex-i.e., from the first word.

When the second word says "inherit" (by means of a 1-bit), there are still two flavors of inheritance. These are First Term inheritance and All Term inheritance. The All Term Variety has been described and is accomplished by using the FIP and CIC POPs. First Term inheritance takes traits only from the first term of the plex--the one closest to the Top of the list from which the plex gets built. This is accomplished by using the FIC and CIF POPs.

The interested reader should examine the plex-definition constants. Get a listing of the assembly LISTDEFS. Turn to the back where the external definitions are and look up SUMPLEX. Then examine the lines around the location of SUMPLEX.

For the mechanism of inheriting traits in the All Terms fashion, look at a routine called Inherit Plex Traits (INPXTRS). You will find in there some techniques for special handling of the Complex Mixture trait (CPXMXFLG) which is discussed in the section on Arithmetic Expression Generation in Chapter 3. INHPXTRS is in the Interpreter (INTERPTR).

1.13.4 Inheritance of Mode

Inheritance of mode works in much the same way as inheritance of traits. The mode field, however, is four bits, considered as a number, whereas the traits are individual bits (20 of them). Basically, mode inheritance works like this: the highest mode of any term in the plex becomes the mode of the plex. But this is an oversimplification.

In the first place, mode may come from the plex itself or from the terms of the plex. (This happens in a similar manner to trait inheritance: the mode field of the second word is either zero, indicating that the mode comes from the first word, or X'F', indicating that it is inherited from the constituents. We do not actually write X'F'; we write "-1", because this will work regardless of the size of the mode field, which was 3 bits on the 9300.) In the second place, even if inheritance comes from the constituents of the plex, it may come from only the first term. This is the case if the plex is built with the FIC or CIF POPs.

In the third and most important case, there is a peculiarity about DOUBLE and COMPLEX. If both these modes occur in the same expression, the mode of the expression is DOUBLE COMPLEX, regardless of whether there are any actual DOUBLE COMPLEX elements in the expression. This quirk is handled automatically by the inheritance mechanism in the plex-building POPs.

In considering the idea of "highest mode wins", the fact that LOGICAL and LABEL are higher than the arithmetic modes is never important. It just happens that these modes are never compared.

In realtional operators, when the two operands are of different modes, the question arises: what mode do we compare them in? The answer is the same as for combining any arithmetic operands. That is, the mode of the relational plex is LOGICAL, but the mode in which the comparison is carried out in generated code is as stated above. The problem is, how do we find out what mode to do the compare in? The answer: build a plex. Just as we are about to generate the code for a relational, we build a Comparison Plex (COMPPLEX). We do not want this plex; we just want its mode. So we build the plex, examine its mode, and throw it away. This is done in the routine Make Modes Agree (FIXMODE) in Pass 2.

Trait inheritance works only up the tree structure. But in a way mode inheritance moves up the tree, then down again. The upward inheritance is done during scanning and plex building, when we determine what the overall mode of an expression is. Then during code generation, this information is passed down again. The reason for this is the convention regarding mixed mode in Xerox Sigma Extended FORTRAN IV. In some algebraic languages that allow mixed mode (Algol, for example), each operation is carried out in the higher mode of its two operands. In Xerox Sigma Extended FORTRAN IV (as in 9300 FORTRAN IV and 900-series FORTRAN II), however, the mixed mode rule is: the result will be as if every operation were done in the highest mode of the expression. This convention has some strange ramifications. The expression 3/4 has the value 0 because integer division truncates. The expression 3/4 + 0.0 has the value 0.75; the presence of the REAL constant 0.0 turns the whole

expression into REAL, so the 3/4 is done as a floating divide, not an integer one.

Now in order for this kind of rule to be implemented, you have to know the mode of the expression before you can generate any part of it. This is one of the reasons why we scan a whole expression before generating code for any part of it.

Inheriting the highest mode "downward" is accomplished in Pass 2 with the MODELIST. This is a list which has on its bottom the mode that we want the whole expression generated in. (This has to be a list instead of a temp because we may have to recurse to evaluate function arguments in their own mode.) There are many different entry points to the arithmetic expression generator (over thirty) whose main difference is how they handle the question of mode: where does the mode for the MODELIST come from (if not already there), and does it get thrown away at the end? It is surprising that so many variations in this area are useful.

It would be instructive for the reader to look up the Plex and Mode constants in LISTDEFS. Turn to the back of the listing where the external definitions are. Look up SUMPLEX and examine the lines near the location shown; this will give you the plex constants. For the mode constants, look up REAL. Thery are all in the same area.

Also, for the upward inheritance mechanism itself, see a routine called Inherit Plex Traits (INHPXTRS) in the Interpreter (INTERPRT).

1.13.5 Indexing By Mode

Mode indexing is a technique of doing different things based on the mode of what you are dealing with. Mode indexing is the whole reason for having the BAM POP. It executes one of a series of POPs in a table, based on the mode of what is in W0. This is particularly useful in code generation (where one might generate an AW, FAS, or FAL instruction depending on whether the mode was INTEGER, REAL, or DOUBLE), but the technique is useful elsewhere.

Besides the BAM POP, there is a machine language subroutine that loads the mode field of W0 into the index register. This routine is called WMODXIML (W0 Mode To XI ML) and is called with a BRL.

Usually, mode indexing is done only with the Arithmetic modes (INTG, SNGL, DOUB, CMPX, and KMPX). The only time the other modes are involved is in the generation of calling sequence op codes. In that case, LOGL and LABL are involved, but not UNKN.

1.14 Print POPs

There are a variety of POPs used for printing on the listing device (LO). These are used for the source listing, the object listing, diagnostics, and the summary. Most of the POPs do not actually print anything; they put information into the print buffer (PRBUFFER). The PRL POP is then used to print the buffer, once all the desired information is put into it. It interfaces with a routine in MERRIMAC to do the actual output.

The print buffer is 28 words (or 112 characters) long. The value of PCHPOS (Print Character Position) determines the character position about to be used. The first character in the buffer is called position zero and is never used. This is so that, if the monitor needs to alter column 1 for vertical format control purposes, nothing will be destroyed. Therefore, PCHPOS is always initialized to the value 1, and all print lines actually begin in the second byte of the buffer. Although the buffer is 112 characters long, we never print beyond column 108 (print position 107). This assures that the listings will fit on an eleven inch paper with some margin for error (such as the lines not beginning precisely at the left edge of the paper). There is, however, no check for this in the print POPs. The compiler is just careful to never go beyond this.

The subroutine INIPRTML (Initialize Print) blanks the buffer and sets PCHPOS to 1. This must be done before any printing is done. It can also be done to throw away a line that has been built up, without actually printing it. For example, Pass 1 constructs a listing image for each source line. If LO is requested, but not LS, the print buffer is output to the scratch file (Pass 2 will send it along to Pass 3) for use in the object listing, but then the buffer is reinitialized without having been printed. Of course, after PRL prints a line, it also reinitializes the print buffer (using INIPRINT, which is the same as INIPRTML, but called from machine language instead of from POP language).

The eleven POPs associated with printing are described below.

PRC PRint Character

The address field is interpreted as a right-justified EBCDIC character, and this character is inserted into the print buffer.

For example,

```
SLASH    EQU  '/'

         PRC  SLASH
```

The EQU defining the character is not actually necessary. All of the character values are defined in SYSTEM FORTCOMP.

PRQ PRint Quote

PRQ inserts a string of characters into the print buffer. The string is a quote (see Quotes, 1.15) enclosed in dollar signs. This POP begins at the

word addressed and scans for the first dollar sign. Then it inserts characters into the print buffer until it encounters a second dollar sign. Thus the count is inherent in the quote, and the quotes can be packed to save space.

For example:

```
            PRQ  QCONSTS
            .
            .
            .
QCONSTS     TEXT 'DE:SCONSTANT'
QTEMPS      TEXT 'S:STEMPS'
            etc.
```

Note that when the quotes are packed together, each TEXT line has a multiple of four characters. A quote cannot have a dollar sign in it. PRC must be used for that.

PAC Print Alphanumeric with Count

The character string beginning at the word addressed is inserted into the print buffer. This is not a quote (surrounded by dollar signs). It is a string that has been created by the compiler, typically a symbol in the Symbol Table. The number of characters (i.e. the count) is contained in EATEMP (the Effective Address Temp). For example:

```
            EAT  8
            PAC  CENTRAL1
```

PAM Print Alphanumeric, Minimum width

The character string beginning at the word addressed is inserted into the print buffer, but with trailing blanks suppressed. EATEMP contains the maximum number of characters to print. In other words, starting from the first byte of the word addressed, PAM prints characters until it either finds a blank or exhausts the count.

Note that this is not a true "trailing blanks" test. PAM terminates on the first blank it encounters; thus the strings that it prints are assumed not to have embedded blanks. Example:

```
            EAT  8
```

PDC Print Decimal with Count

EATEMP contains a value that determines the size of the print buffer field. The word addressed by PDC is converted to decimal and then right-justified in that field. Leading zero are not printed. Nor are minus signs (values are treated as unsigned integers). Also, PDC is not designed to truncate digits off the left end of a number. If used with a number that will not fit in the field width specified, PDC will extend the field in order to print the number correctly (i.e. it acts like PDM, below). The compiler takes care not to do this. Example:

```
EAT  5

PDC  SIZE
```

PDM Print Decimal, Minimum Width

The word addressed is converted to decimal and inserted into the print buffer, using as many character positions as are required to express the value with no preceding blanks or zeros. The value zero is printed as the single character "0". Negative values are not provided for; the sign bit will be treated as another bit of significance at the high-order end of the word. Example:

```
PRC  LPAREN

PDM  BLKSIZE

PRQ  QWORDS          where QWORDS = $ WORDS) $
```

might produce the following string on the listing:

```
(43 WORDS)
```

PDM works much like widthless I format at run-time, except that it does not output a trailing blank (or handle negative values).

PHC Print Hexadecimal with Count

EATEMP contains a value that determines the size of the print buffer field. The word addressed by PHC is converted to hexadecimal and then right-justified in that field. Besides working with hexadecimal, PHC differs from PDC in two other ways: preceding zeros are printed; and digits will be truncated off the left end if the field is too small. The latter is necessary so that, for example, the address field of a word can be printed without worrying about the fact that the op-code and register fields are non-zero.

For example:

```
EAT  5 PHC LOC
```

might produce the five characters

```
003A8
```

There is no POP for printing hexadecimal with minimum width.

PBL Print BLanks

This works exactly like X format at run-time. Rather than actually printing blanks, it just moves the print position by incrementing PCHPOS. (The print buffer is initially blanked.) The immediate address of the PBL determines the number of positions to move (or blanks to print). This number may be negative, so that PBL may be used to back up. If this is done, subsequent moving forward with PBL will not erase previously created information. Again as with X format, attempting to back up beyond the first usable character position will only cause backing up to the first character position.

Example:

```
PBL  5          Space five positions
```

TAB TAB to print column

TAB sets the print position (PCHPOS) to the value specified by the effective address of the POP. This should not be beyond position 108. TAB cannot be used to back up. Attempting to TAB backwards results instead in spacing forward two positions (i.e. a PBL 2). The reason for this is that the compiler happens not to need the ability to TAB backwards (it can back up using a negative PBL). However, the situation sometimes occurs that an operand on the object listing is too long to fit in columns 19-36 and flows over into the comments field.

Rather than have to test for this situation or back up blindly and print the comment on top of the operand, the TAB POP automatically moves the comment over two spaces. It is not really critical what columns those things go into.

PRL PRint Line(s)

This is the POP that actually outputs the print line built up by the other print POPs. If the effective address (n) has the value 1, the current line is printed on the LO device and the print buffer is reinitialized to blanks and PCHPOS to 1. If n is greater than 1, the current line is output, followed by n-1 blank lines, for a total of n lines.

For example:

PRL 3

would print the current line and then two blank lines.

If n is negative, PRL does a page eject. This may be accomplished by writing PRL PAGE. PAGE is defined with a negative value in SYSTEM FORTCOMP.

PRL also does some testing to avoid printing extra blank lines at the top of a page. If there are only m lines remaining on the page, a PRL m+k will act just like a PRL m. It will not print the extra k blank lines at the top of the next page. This assumes that the operating system is able to tell NPRLINES (in MERRIMAC) how many lines are left on the LO page. Some can and some cannot; See 4.3.7.

GNL Guarantee N Lines on LO page

The effective address determines the value of n. If there are fewer than n lines remaining on the current page of the LO file, the page is ejected. Otherwise, no action is taken. GNL is used to assure that certain items will be printed on the same page. For example, headings and the tables they head, source lines and their diagnostic markers, and the whole program size table at the end of the summary. Since we typically do not know the size of a table when we start it, we guarantee enough lines to get the heading and the first line of contents. Certain operating systems do not provide this information, in which case the POP does nothing; see 4.3.7.

1.15 Quotes

Quotes are strings of alphanumeric characters within the compiler. They are used for two purposes:

1. Input scanning, with the QSA POP. To compare a given quote with the next N characters in the source input string. For example, are the next eight characters "FUNCTION"?

2. Printing of messages on the source or object listing, with the PRQ POP. This is used particularly by the summary and by some diagnostics.

Quotes are stored four characters per word, enclosed in dollar signs. The next quote begins immediately following the dollar sign, even though this may be in the middle of a word. The quote POPs automatically scan until they find a dollar sign and then begin. When they find a second dollar sign, they terminate. A quote may be of any length, and may occupy a number of consecutive words. Obviously, a quote cannot contain a dollar sign. At no

time does the compiler need such a thing. It can always print a single dollar sign (e.g. after a global label) using the PRC POP (see below). The assembly label attached to quotes always begins with the letter "Q".

1.16 Terminology

The following are terms used rather often in this manual as well as in the comments in the compiler listing.

Flag A flag is usually a trait, which is a one-bit quantity associated with a pointer. Traits are tested with the SOF POP (Set On Flage). In a few cases "flag" is used to mean "trigger".

Gen Gen is short for "generate" and means to generate object code. It is found almost exclusively in Pass 2.

Inheriting Inheriting is a process that is carried on by the Plex-building POPs so that the top of the tree may know what is going on at the bottom. Both traits and mode can be inherited. For a detailed explanation, see Sections 1.13.3 and 1.13.4.

Komplex Komplex is a term for DOUBLE COMPLEX. We used it because we needed a single letter that could stand for this mode. It appears in many labels as an abbreviation.

List A list is a variable-sized data structure. Lists are used quite extensively throughout the compiler, and there are 85 different lists at the current time. See sections 1.8.1 and 1.4. For information on specific lists, see section 1.5.

Mode Mode is the word we generally use in the compiler for "type". This can be INTEGER, REAL, DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX, or LOGICAL. We also have modes called LABEL and UNKNOWN. See section 1.13.

Plex A plex is a data structure useful for representing expressions. It consists of a plex pointer (usually on the bottom of the WORKLIST) and some data on the PLEXLIST. See all of section 1.10 for details.

Pointer A pointer is a one-word item that points to some data on a list. Generally, there are regular list pointers, Symbol Table pointers and plex pointers. See sections 1.4.4, 1.11, 1.12 and 1.7.2.9.

POP The word POP is short for Programmer OPerator and is an insturction in the interpretive language in which the compiler is written. See sections 1.6, 1.7 and 1.8.

Prologue This is the code that is generated at an entry point of a subprogram to set for dummy addressing. See 3.78.3.3.

Range — A range is a pair of numbers--a lower and an upper--against which something may be tested to see if it falls in that range--i.e., whether it is greater than or equal to the first and less than or equal to the second. The concept applies to the SOR POP, the SRC POP and the QIX POP, whose descriptions can be found in section 1.7.

Register — This word is usually used as a verb and means to enter something on a list in such a way that it will not appear on the list twice. This means, search the list to find if it is already there; if it is, return a pointer to it; if not, create a new item on the bottom of the list and return a pointer to it. Registering is used with identifiers, labels, constants and so forth.

Rung — A rung is a particular word in an item on a list. For example, the SUMRYLST HAS TWO_WORD ITEMS: THE first word is called the Address Rung (YADDRESS) and the second Symbol Pointer Rung (YSYMPNTR). Rung numbers are important for addressing specific words in items on lists. Symbols for these rung numbers almost always start with the letter Y. These have values indicating displacement from the top of the item, so YADDRESS EQU 0 while YSYMPNTR EQU 1. The rung names are shown in diagrams accompanying many of the writeups for lists in Section 1.5.

Scan — Scanning is the process of looking at source lines to try to figure out what they mean. This is all done in Pass 1 and is abetted by several POPs which are described in Section 1.7.2.13. Detaisl fo how particular statements are scanned are given in the throughviews in chapter 3. See also sections 2.2, 2.9 and 2.10.

Temp — A temp is a memory cell in the object program that is used for temporarily storing intermediate results. Ordinary temps are shown on the object listing as 1TEMP, 2TEMP, and so forth; these temps are shared by many statements. There are also private temps (1PTMP, 2PTMP and so forth) that are not shared, and there is the real-time temp (1RTMP). Temps are allocated in the same control section as local variables.

Trait — There are twenty different traits associated with pointers. Each is a one-bit quantity that is either there or not there. They are discussed in section 1.13. This word is usually synonymous with "flag".

Trigger — A trigger is a one-word temp within the compiler that contains a logical value TRUE or FALSE. They are set with the MEF and MET POPs and tested with TMT. They are used for all sorts of things.

Variable — A variable is that which can appear on the left side of an equal sign in an assignment statement. It is a scalar or an array element or a multiple dummy element.

1.17 Assembly Listing Conventions

This section discusses some of the conventions used in the preparation of assemblies (of the various parts of the compiler) in order to increase the readability of the listings. It is by no means true that these conventions have been completely adhered to, either during the original project or during subsequent maintenance. However, some of them have been followed fairly consistently, so it is worth understanding them. Also, it is hoped that future work will follow the conventions as much as possible.

1.17.1 Title cards

Every listing begins with a comment card of the form

```
* XDS SIGMA 5/7 FORTRAN IV - SUMMARY      (FPASS3)   8/14/70
```

The name of the deck begins in column 30 and corresponds to the name of the primary DEF (see below).

The name in parentheses indicates the overlay segment that the assembly belongs in. (For an explanation of names like FPASS3, see Overlay Structure Details, 4.5.18.) The date is always changed when new assemblies are made, as it is the only thing that keeps track of which binary and compressed decks go with which listings. This means that if two reassemblies are made on the same day, we push the date on the second one up to the following day. It is more important to correctly identify the version than to know the actual day the assembly was run.

If the title has to change (e.g. XDS to Xerox, or 5/7 to 5/9. or Sigma to Enigma), this should be done consistently on all assemblies.

1.17.2 REFs and DEFs

All the REFs and DEFs are listed in alphabetic order at the end of the listing, in two sections headed EXTERNAL REFS and EXTERNAL DEFS.

The REFs come first. Each REF is on a separate line and shows not only the name of the symbol being REFed, but, beginning in column 37, the name of the assembly in which that symbol is DEFed. This makes it much easier to find things in other assemblies that are referenced in the current assembly. This list must be kept current if any of the DEFs are moved to another assembly, even though it means reassembling a program that otherwise needs no changes.

the DEFs are also in alphabetic order except for the DEF of the assembly name, which comes first and is split off from the others by a blank comment card. Each symbol should have both a DEF and a "Z SET", to get the assembler to print the location at which the symbol is defined. The comments field of the DEF line should contain a brief explanation or definition of the symbol. As an example, to show how this works in conjunction with the REF conventions: OUTPROG calls EVALPNTR. OUTPROG's REF list indicates (in alphabetic order) that EVALPNTR is in PASS3. PASS3's DEF list shows (also in alphabetic order) that EVALPNTR is defined at location 00187 and stands for Evaluate Pointer.

The first DEF, which is the offical name of the program, is the same as the name that appeared on the title card and is always defined at relative location 00000 of the assembly. This is mandatory for the compiler debug package to interpret addresses properly. Note that there is also an additional DEF at the front of the listings on all the assemblies that do not lie in the root segment. This is the name preceded by # and is used for system patching in the absence of the compiler debug package. This is explained in Overlay Structure Details, 4.5.18.

If there are large numbers of conditional REFs or DEFs, these may be separated, but should still be in alphabetic order themselves.

1.17.3 Local Labels

The use of local labels helps document which labels are really important in a section of code, i.e. which ones can be referenced from outside. It is always safe, then, to change around or remove local labels without having to worry about references that might exist in other parts of the program.

Local labels have the form nAm, where A is an arbitrary letter that is used throughout the program, n changes between sections of code, and m changes within a section of code. Thus, one section might use 5Z1 and 5Z2, 5Z3, and 5Z4. The next section might use 6Z1 and 6Z2. Local labels should not be referenced outside their own section. In other words' once a 6Z label has been used, there should be no references to 5Z labels. A convention observed in some parts of the compiler (particularly in Pass 2) is to end the local label sections explicitly. This is done with a comment card having an 'op code' EL. That is, column 1 is an asterisk, and columns 10 and 11 say "EL"; the operand field is the name of the label section that is being ended, such as 5Z. It means that no further labels will be defined after this point that start with the characters 5Z. Ideally, it should also mean that there will be no references beyond this point to labels beginning with 5Z, and it usually does mean this.

To keep track of what values of n have been used, the label field of the END card should contain the highest section of local labels used. Thus if the END line says

```
23Z         END
```

this means that local labels up through 23Z have been used, but it is safe to use labels beginning 24Z in a new section of code. Obviously, the END line must be kept up-to-date.

Relative addressing (e.g. $+2) should <u>never</u> be used. If a line needs to be referenced, it should have a label.

1.17.4 Other labels

Other (global) labels do not begin with a digit, so that they will not be confused with local labels (or with system library routines, most of which begin with digits). All the labels are eight characters or less and make some attempt to be mnemonically descriptive.

There are also certain classes of symbols that generally begin with (or end with) the same letter or letters. For example, quotes begin with "Q", error messages with "ER", and list rung values with "Y". Machine language subroutines called from POP language (i.e. with BRL) end with "ML".

1.17.5 Assembly columns

Due to the fact that large portions of the compiler were copied from 9300 FORTRAN IV, and to the fact that various coding columns have been used both on the 9300 (the compiler was unable to use the standard columns because they did not provide for eight-character labels) and on Sigma (Sigmet differed from Meta-Symbol), parts of the compiler unfortunately have code that wavers back and forth in different columns. This is particularly true in Pass 1, which contains the least machine-dependent (and therefore most heavily copied) code. However, all new code uses the standard Meta-Symbol columns, 10, 19, and 37. Indirect references are offset one column left so that the asterisks stand out in column 18. Also, comments on comment cards (especially headings) are offset somewhat to the left of these standard columns so that they also stand out.

1.17.6 Comments

Each distinct section of code should have comment cards preceding it that tell what it is and what it does. Also, instead of commenting each instruction line individually, the compiler often uses multi-line comments to explain what is really being done. For example, instead of :

```
        FET     PTMPCNT     GET  PTEMP COUNT
        ETW     =1          EXTRACT LOW-ORDER BIT
        SGD     =0          WAS THAT BIT PRESENT?
        JAF     9B4         NO, DO NORMAL TEMPS
```

the following is used:

```
        FET     PTMPCNT     IF NUMBER OF P TEMPS IS EVEN,
        ETW     =1              PUTTING THEM IN FIRST
        SGD     =0              WON'T HELP ANY, SO
```

```
JAF      9B4                GO DO THE NORMAL TEMPS FIRST.
```

Note that the subsequent lines are indented to show that they are part of a multi-line comment. Care should be taken in changing code within the compiler not to disrupt multiline comments.

One other thing: many of the comments are followed by question marks. Unfortunnately the question mark does not print on most printers. When updating code, the question marks should be retained on comments that obviously should have them. Then when listings are made on printers with fuller character sets, the question marks will appear.

1.17.7 SYSTEM directives

Instead of always using SYSTEM SIG7FDP, we restrict the system call so that it documents whether the assembly actually uses any Sigma 7 only, floating point, decimal, or privileged instructions. The entire FORTRAN system (including library) uses no Sigma 7, decimal, or privileged instructions. The compiler itself also uses no floating point instructions, as they are not really very necessary. Therefore, all of the compiler assemblies begin with SYSTEM SIG5, unless they use no machine language at all (only POPs), in which case they use only SYSTEM FORTCOMP.

CHAPTER 2. SPECIFIC AREAS

Specific areas that are mostly within one pass or another, but are associated with no particular statements.

2.1 Compiler Initialization

The compiler is designed to be completely self-reinitializing. It does not depend on loading for correct operation. It can be restarted at any time by branching to its starting point (FORTRAN4). The initialization is done in two principal places: in the assembly FORTRAN4 and in PASS1. Some of it is done for each successive compilation. Some is done only once each time the compiler is called into memory to begin operation. These two areas are discussed below.

2.1.1 Initialization done only once

1. Call INITIAL in MERRIMAC to initialize the monitor interface section. This mainly consists of opening files, and is discussed under Open Files in Section 4.4.1.

2. Call GETLMEM in MERRIMAC to get list memory (working storage). This is described under Get List Memory in Section 4.3.1.

3. If this is a debug version of the compiler, call INIDEBUG to initialize the compiler debug package.

4. Set the FIRSTime trigger to TRUE. This is described under Interpret Control Card Options, in Section 4.3.3.

2.1.2 Initialization done before each program compiled

1. Call INILISTS in INTERPTR to initialize the lists. This is described under Dynamic Lists in Section 1.4.8.

2. Call FOPTIONS in MERRIMAC to interpret the control card options and set up the corresponding triggers. See Interpret Control Card Options, in Section 4.3.3.

3. Initialize the triggers that control the use of the scratch files.

4. Initialize all the other counts (such as character counts, number of error messages, etc.) and triggers (such as Main Program trigger, NAMELIST Trigger). These are gathered together in the section called COMPINIT in PASS1F.

5. Call INIPRTML in INTERPTR to initialize the print POPs to start at the beginning of a new line.

6. Call INICTTML in PASS1ML to initialize the Character Translate Table. This is described under Character Translate Table, in Section 2.4.

2.2 Character Scanning

Most of the input scanning done by the compiler proceeds as if there were a continuous string of characters coming in from the outside world. The fact that this input is divided up into separate cards, some with continuation cards, is of little importance to most of the compiler. Those details are handled by the lower level routines. With only one exception, the compiler gets all of its characters by using the following five POPs:

CSA	Character Scan with Answer
CSK	Character Scan and Keep
CSF	Character Scan or Fail
SOC	Set On character Class
QSA	Quote Scan with Answer

The one exception is Symbol Scan, which is in machine language and goes directly to the lower level routines to get its characters, since a symbol is considered a cohesive unit, much like a quote.

Before discussing these POPs and the lower level routines they call, it is a good idea to understand how the input string is stored and how characters are accessed. As input cards are read in, they are stored, one after another, in the Input List. Originally, each card was considered to be 80 characters long, of which 72 characters are of interst to the compiler. Recently, however, an extra word has been tacked on the end of each card to keep track of line numbers when doing compressed updates (i.e. under the XOS monitor). Therefore, the Input List looks as if each card were 84 characters long, the first 72 of which are important. Every character in the input string is numbered sequentially, beginning at zero (for the first character on the first card) and running on up through the last card. Thus each character can be identified by a unique number. This number is called the Input Character Count (INPCHCT). It is used not only by the scanning POPs, but also when attaching error markers to certain characters in a source line. At any moment, the current value of the Input Character Count determines which character is being scanned.

In conjunction with the Input Character Count, a word called Current Character (CURRCHAR) tells whether the character in question has been used yet. This requires some explanation. Initially, CURRCHAR is zero and INPCHCT is -1. When CURRCHAR is zero, this indicates that the "current" character (the one pointed to by INPCHCT) has been used, and we need to get a new one. Thus the first time we ask for a character, INPCHCT will be incremented to zero and CURRCHAR will contain the full word representation of the character that appeared in column 1 of the first card. (The full-word representation is that taken from the Character Translate Table, which is described in 2.4.) If we now do a non-moving POP such as CSK or SOC, the POP will pick up CURRCHAR and examine it, but it will not update either CURRCHAR or INPCHCT. When we do a

POP that actually moves past a character, such as CSA (when it finds the requested character), the POP zeroes CURRCHAR but does not update INPCHCT. The latter happens only when we ask for another character. When we do that, the POP used will find that CURRCHAR is empty (zero), so will increment INPCHCT, get the new character, and put it into CURRCHAR. There are two reasons why we use this "two stage" method of getting characters, rather than just incrementing INPCHCT each time we are through with a character:

1. It is essential not to read even one character beyond where the compiler has specifically asked to be. Doing so may take us to a new continuation card, and we do not want to do that until we really need what may be on it. For example, once we have read the characters "END", we must terminte without reading another card to see if it is a continuation card.

2. For the purposes of diagnostics (e.g. the ERC POP), the "current" character may be either the one we are now looking at and have not used or the one we just used if we have not looked at another one yet.

There is also a count called the Last Active Character Count (LACTCHCT). Every time INPCHCT is updated to point to a new active character ("active" usually means non-blank), LACTCHCT receives the old INPCHCT. This enables us to put error markers on "previous" characters after we have already gone off and looked at another character. For example. suppose that ABC has been declared LOGICAL type and N INTEGER type. Then in the statement

```
N = ABC + 3
```

the compiler is not sure that the name is really ABC (it might turn out to be ABCD) until it has seen the plus sign. At that point it wants to back over the intervening blanks and put its marker under the letter C.

All of the character scanning POPs call Ready Scan Character (RDYSCNCH) to get the current character. (See the individual POP write-ups for descriptions of these POPs.) If they want to step past the character (as CSA does, for example, when it finds the character requested), they simply zero CURRCHAR. This signals that the character has been used. When several characters are going to be stepped through, one can do this either by zeroing CURRCHAR each time (as the OSA POP does) or by calling Next Active Character, which gets you the next character (NXTACTCH), regardless of whether the current one has been used. All of these lower level routines, described below, are coded in machine-language and are called from machine-language code via BALs.

2.2.1 Ready Scan Character RDYSCNCH

This routine returns in register R (i.e. 8) the contents of CURRCHAR, unless that is zero, in which case it calls Next Active Character (NXTACTCH) and returns what NXTACTCH sends back in R. In other words, if the current character has not been used yet, keep returning it. Once it has been used, step to the next character. (The stepping, however, is done by NXTACTCH.)

2.2.2 Next Active Character NXTACTCH

Returns in R the next active character in the input string. Blanks are not considered active unless the USEBLTRG is set (this is done by Next Hollerith Character, when scanning alphanumeric fields, for example, and also by SINCOLM1 when scanning in-line assembly language). This routine also has to handle continuation cards, if it is appropriate to do so.

First we call Next Input Character (NXTINPCH), which gets us the absolute next character in the input string. If that character is a blank, we return it if USEBLTRG is set; otherwise we go back and call NXTINPCH again. If NXTINPCH gets to column 73, it returns a special character called ENDCHAR. If we get one of these we have to check for continuation, which is described below. Assuming it was not one of these, and we want to return the character (which may be a blank, as above), we are essentially done. NXTINPCH has taken care of updating INPCHCT and storing the cahracter into CURRCHAR. However, if the SVACTRIG (Save Active Characters Trigger) is set, we have to add this character to the bottom of the Alpha List. This is a list used by the FORMAT and OUTPUT statements to accumulate all the active characters, while doing other scanning, and then pack them up and send them on to become part of the generated code. This is described in "Saving Alphanumeric Characters", 2.5.

When an ENDCHAR is returned by NXTINPCH, there are a number of tests we have to make to see if continutation is imminent. Continuation is not allowed for lines containing "S" in column 1. This is indicated by the SCARDTRG. Lines beginning "END" also do not allow continuation (indicated when NOTENDTR is FALSE). In either of these cases, the ENDCHAR becomes active. Otherwise, we have to look at the next card. When NXTINPCH got to column 73, it automatically moved the INPCHCT up to point to column 1 of the next card (although that card is not actually read until we ask to look at that character). So we call NXTINPCH and look at column 1. If we are on-line, and the current card did <u>not</u> have a colon at the end, then continuation is not allowed and NXTINPCH will send the ENDCHAR back again. In this case too, the ENDCHAR becomes active. Next we check whether the character in column 1 of the next card is an X. If it is, we check the XTRIG to see if the X option was specified; if it was not, than this cannot be a continutation card and the ENDCHAR is considered active; if X was specified, then we proceed as if the X had not been in column 1. The procedure then (or if there was no X in column 1) is to check columns 1 through 5 to make sure that they each contain either blank or a digit. If any one of them is not one of these, this cannot be a continuation card, and the ENDCHAR becomes active. If columns 1-5 are valid, then column 6 must contain a proper continuation character (indicated by the CONTFLAG bit in the character translate table word). If it does not, then the ENDCHAR becomes active. If it does, then the this a continuation card and we want to start looking for an active character in column 7. In this case, we zero the NOCONTRG (No Continuation Trigger) to indicate that there has been a continuation card. (The NOCONTRG is described under "Card and Line Handling", 2.3.)

If any of the tests failed, and this is found to <u>not</u> be a continuation card, then the ENDCHAR must become active. This means we have to do three things.

First we put the INPCHCT back to point to column 1 of this card, so that when we later get around to scanning this card, we will start at the right place. Second, we store ENDCHAR in CURRCHAR. And finally, we negate the NOCONTRG. If there have not been any continuation cards so far on this statement (indicated by NOCONTRG=1), this will cause NOCONTRG to become negative, meaning there are absolutely no continuation cards on this statement. If, however, NOCONTRG has already been zeroed by a previous continuation card, negating it will have no effect, it will remain zero and indicate that ther has been at least one continuation card.

2.2.3 Next Input Character NXTINPCH

Most of what this does has already been described under Next Active Character. In general, it returns the next character (Of any kind) from the Input List, increments NXTINPCH by one, and stores the character (in its Character Translate Table full0word representation) into CURRCHAR. If the Input List has run empty, it reads another card, unless the CONTTRIG is set, which indicates that we are on-line and no colon was found (by SIREAD), so there cannot be continuation; in this case, the ENDCHAR is returned. Note also that reading another card requires 21 words on the Input List. If sufficient room does not exist, the list will have to be reassigned. NXTINPCH also appends the line count in the 21st word of the image. This is described under "Card and Line Handling", section 2.3.

Assuming that there is a character in the Input List that we can get, we have to find out if it is beyond column 72. We do this by dividing the INPCHCT by 84 and looking at the remainder. If it is greater than 71 (column 1 has remainder zero), we advance INPCHCT to column 1 of the next card (but we do not read in the next card yet; in fact, the INPCHCT may point us beyond the end of the Input List, but we will find that out next time in), and return an ENDCHAR, to let NXTACTCH decide what it wants to do about that.

If we are not beyond column 72, we go get the character, then get its Character Translate Table representation (see Character Translate Table, 2.4), which we store in CURRCHAR and also return in R.

"Getting" the character pointed to by INPCHCT, and determining (above) whether or not we are off the end of the Input List, require two additional counts (besides INPCHCT), which are described below.

The Input Stop Count (INPSTCT) is the input character count of the last character in the Input List at any given time. Thus, when INPCHCT becomes greater than INPSTCT, the desired character is not in the list, and another card must be read.

The Input Offset (INPOFSET) is used in combination with the Input Character Count to determine the exact position in memory of any character. The INPOFSET is essentially the byte address of input character number zero. If this is added to the INPCHCT, the result is the byte location of the desired character. Initially, of course, INPOFSET is the byte address of the beginning of the first word in the Input List. As cards are added to the

bottom of the Input List and old ones are taken off the top, INPOFSET does not change. Even though character zero is no longer present, the other characters are still in the correct positions relative to where it would be if it were still there. However, whenever the Input List is reassigned, INPOFSET must be adjusted. (This is handled by Reassign Memory.) Normally the Input List is reassigned downwards (towards lower memory), so INPOFSET decreases; it may even become negative if the Input List is reassigned downwards enough times.

2.3 Card and Line Handling

The terminology used in the compiler for cards, lines, statements, and compound statements is a little confusing and not entirely consistent. Compound statements, in particular, have complicated the situation (they were not designed into the system originally, on the 9300) because not only can a statement have several cards (continuation), but a card can have several statements (compound). Or several cards can have several statements, but all connected by continuation cards. That whole unit we generally call a line. (This is, admittedly, a poor choice, since continuation cards are now often called continuation lines in the literature, and since input from a typewriter is reasonably thought of as being broken into lines, not cards. But we have to make do with the terminology that is in the listings.) Consider the following example:

```
      X = Y ;
C PRINT 5, X ; K = K - 1 ;
C GO TO 6
```

There is one line, three cards, four statements, and one compound statement. Sometimes, however, when we refer to "a statement", we mean the whole thing, up to the end of the continuation cards, even if it is a compound statement. We will try to avoid the use of the term "line" except where absolutely necessary.

Input card handling falls into two sections: reading in cards and appending them to the Input List; and printing out and disposing of cards, thus shortening the Input List. Note that the printing out is not done at the time we input a card; it is done when we are all through with the whole line (i.e. including continuation cards).

All source input cards are put onto the Input List during Pass 1. (Pass 2 uses the Input List for the output from Pass 1. See Pass1/Pass 2 Interface.) Since it is a dynamic list, any number of continuation cards may be used, at least until memory is used up. The routine that actually reads a card and puts it on the Input List is Next Input character, NXTINPCH (see Character Scanning). It does so whenever it is asked for a character and the Input

Character Count points beyond the last character in the Input List. Initially, of course, there are no characters in the Input List, so the first time the compiler needs one (which is a SOC DIGITFLG in Label Field Scan), NXTINPCH will have to read a card. Note that it never reads ahead, never reads a card until it is specifically asked for a character it does not have. This is necessary for such things as the END statement.

When NXTINPCH wants to read a new card, it must make space for 21 words on the Input List. If there is insufficient space, it calls Reassign Memory to make that much space. The 21st word of each card is not actually read in. It is the line count, a feature that has been added to handle the updating facilities desired under XOS. On other systems, this line count is a simple integer count, determined by dividing the Input Character Count by 84 and adding 1. Were it not for this XOS facility, this count would not be needed, because the value LINECNT is continually incremented by 1 to count the lines. (And here we do not really mean lines, we mean cards. This terminology is distinctly a problem.) See also "Line Numbers", 3.14.

When NXTINPCH comes to column 73, it sends back an ENDCHAR, and moves the INPCHCT up to column 1 of the next card, but does not read it yet. The next higher routine, Next Active Character, makes the decision whether to look for a continuation card. If we have an S in column 1 line (indicated by SCARDTRG) or an END statement (indicated by NOTENDTR being FALSE), NXTACTCH will refuse to look for a continuation card. Even if it does look for one, NXTINPCH may refuse to give it one if the CONTTRIG is set. This occurs only on-line, when a source line does not end with a colon, which means that a continuation line is not allowed.

Assuming that NXTINPCH does read a continuation card, it tacks it on to the Input List along with the original card (or cards). A couple of interesting situations can occur. It is possible to scan into a continuation card (or several continuation cards), and then find out that this is a different kind of statement than you though it was, so you have to back up and rescan. This is no problem, since all the original cards are still on the Input List. This second time, when we come to the end of the first card, however, it will not be necessary to read again, because NXTINPCH knows that the continuation card is already theree (since ther is still more information on the Input List). Normally, scanning continues until we reach a non-continuation card, which is read in, but not used when we find that it is not a continuation card. It stays around until the next statement is scanned. However, it is also possible to have a serious error in the middle of a statement, such that scanning is discontinued and we do not know whether there may yet be continuation cards following. This is handled by Release Line, described below.

When Statement Scan is satisfied that it has finished with this statement (or compound statement, or line), it returns to the overall control of Pass 1 in the assembly PASS1F, which calls the routine Release Line (RELLINE) to dispose of the line.

2.3.1 Release Line (RELLINE)

This routine releases one full statement (or compound statement). This may be a single card or it may include continuation cards, which may or may not have been scanned (or even read in) yet. (Usually they have been, but if the statement fail partway through, they may not have.) This involves printing it (if necessary) along with any error messages, sending it to Pass 3 to appear in the object listing (if LO was requested), and in any case, removing it from the top of the Input List.

First, if there are any errors on this statement (as indicated by the presence of something on the Error List), increment the Number of Statements With Errors (NRSTER) count and set the PRINTRIG to force this line to be printed. Then SORT and FLIP the Error List, so that the error messages will be sorted into order by character position, with the lowest at the bottom of the list. This is so that we can MOF them from the bottom of the list and get them in forward order through the statement.

Now call Release Card (RELCARD), which lops one card off the Input List, after having printed it (if necessary) along with any error messages that pertained to that particular card. Now, if there were no error messages, check whether there is a continuation card following, and if so, repeat the process of calling RELCARD until no more continuation cards are found. This process of checking for continuation cards needs some explanation.

The NOCONTRG (No Continuation Trigger) is initialized at the beginning of each line to the value 1. If, the first time Next Active Character looks for a continuation card, it does not find one, it negates NOCONTRG. Thus a negative (TRUE) value of NOCONTRG tells RELLINE that there definitely is no continuation. If the scanning fails part way through the first card, and we do not know whether there is another continuation card yet, then NOCONTRG is still positive (FALSE), which means "we don't know whether there is continuation". Also, if NXTACTCH does find a continuation card the first time it looks, then it sets NOCONTRG to zero. Subsequently finding that there are no more continuation cards will negate NOCONTRG, but since it has previously been set to zero, negating it will have no effect. In this case, NOCONTRG will still be positive (FALSE) which indicates "there may be continuation cards on this statement, but I do not know exactly how many, so you had better assume that we do not know whether there is continuation".

What all of this means is that, in the common case, one statement on one card (and the next card is not a continuation card), NOCONTRG will be negative and tell RELLINE that there definitely is not continuation. In all other cases, we have to check for continuation specifically. This is done by NOCONTST (No Continuation Test), which returns the reverse of the answer to whether there is any continuation. NOCONTST assumes that the Input Character Count points at column 80 of the current card and Current Character (CURRCHAR) is zero. Both of these conditions are set by Release Input (RELINPUT), which was called by RELCARD. NOCONTST saves the INPCHCT and does any old scan POP that will ask to look at the next character. If doing this causes INPCHCT to change, then the low level character routines must have found a continuation card and

we are now pointing at column seven of the new card. We are not really interested in column seven of the next card, but this served the purpose of finding out whether there was any continuation. On the other hand, if the low level routines did not find continuation, INPCHCT will still be on the current card.

If RELCARD does produce error messages on a particular card, the procedure is the same except that an extra blank line is put out to separate the last error message from the next continuation card.

After the last card of the line, we test whether the statement was deleted as indicated by the DELTRIG, set by Illegal Statement (ILLEGSTA). If not, this is normal, so we turn off the CONTTRIG and exit. Otherwise, we print "STATEMENT DELETED" and a blank line, increment the number of statements deleted (NRSTDEL), and then turn off the CONTTRIG and exit.

2.3.2 Release Card RELCARD

Releases one card image (i.e. one input record) from the Input List. Also prints and/or outputs it to Pass 3, along with the error messages that pertain to this card (if any).

If this is the first card in the program, then we want to list it even if LS has not been requested, unless NS has been requested (see PRIFSUM, which sets the PRINTRIG if this is so). See also Label Field Scan (2.9.5) for an explanation of why LINECNT sometimes is zero.

If neither LS nor LO has been requested, and the PRINTRIG (Print Trigger) is not set (meaning first or last line, entry point, or errors have been detected), then there is no reason to build a print image, so we just release this one card from the top of the Input List (see Release Input, RELINPUT, and exit false to tell RELLINE that there were no errors.

Otherwise, we build a print image of this card. If there are going to be any errors, we guarantee three lines on the page. This may not do any good, because there may be a number of errors, or they may be on later continuation cards, but in a lot of cases, this will get the card, the marker, and the message on the same page. The first thing to go into the print image is the line number. This is supposed to be followed by a colon and one space and then the 80-character source image. However, some changes have been made to this for XOS (see Line Numbers, 3.14).

If the PRINTRIG is set (meaning special line), we also print it on the DO (Diagnostic Output) device (see PRNTDOML). On the other hand, what a difference it makes to put in a new ribbon if SI and LO are the same device (presumably a typewriter), we do not bother to print the source listing, since the user has one that he just typed.

In this case, instead of calling SENDPRBF (Send Print Buffer), we just call PBOUT (Print Buffer Output to Pass 3) if LO was requested, and then reinitialize the print buffer.

Then we go to SENDERRS and check to see if there is anything on the Error List. If not, we exit back to RELLINE. Otherwise, we have to process the errors (if any) that pertain to this card. We reserve the Work List and go into a loop, pulling the error message entries off the Error List and looking at the character count for each. If the character count is greater than the last column on this card, then it pertains to the next card, so we put it back on the Error List (until the next time RELCARD is called) and finish with this card. For the error messages that do pertain to this card, we pile the message pointers up on the Work List, and use the character count to locate the position in the buffer where we ought to print an error mmarker (vertical bar). Note that the PBL LEFTMARG is used to keep backing up to the left margin so that the character count will move us over to the proper position for the marker. It is not possible to TAB backward (see the TAB POP); the only way to do it is to print a negative number of blanks. The PBL POP will automatically put you at the left margin if you try to back up too far.

If we encounter an error whose character position is the same as the one we just had, we have to save it on the SVERRLST so t that we can go back and print a second line of markers (or as many as necessary).

When we either run out of errors on the Error List or encounter one that belongs on the next card, we check to see if there were any errors on this card. If not, we exit false to RELLINE. Otherwise, we go through the same set of checks about where we ought to list this line (the one containing the markers). Furthermore, if LO and SI are both on the typewriter, we have not actually printed the source line; the user typed it in. This puts it in a different position on the page than it would be in if we had printed it out. Therefore, to get the markers to line up under the proper characters, we have to move the print buffer around.

If there were any double markers (one the same character), we FLIP the SVERRLST (to get it back into the proper order) and copy it back onto the ERRORLST, then go back and repeat this whole procedure until there is nothing on the SVERRLST.

Now that all the markers have been put out (possibly on several lines), there exists, on the Work List, pointers to the messages that go with those markers, and those pointers are in precisely the order we want, so we just pull them off, one at a time, and call PRERRMSG (Print Error Message) to print them out (and/or send them to Pass 3).

2.3.3 Send Print Buffer SENDPRBF)

If LO has been specified, call PBOUT to send the print buffer to Pass 3. Then, if LS has been requested, or if this is a special line (PRINTRIG set), print the current line on LO (with a PRL POP). If not, then just throw away the current line by reinitializing the print buffer to be ready for a new line.

2.3.4 Output Print Buffer (PBOUT)

File a SOURCENO on the Code List, and then move it right away over to the Output List (see Code List manipulation in Pass 1). Follow this with a word indicating the size of the print buffer, and then the whole print buffer itself.

2.3.5 Release Input (RELINPUT)

Add 21 (NINPWDS) to the Base, Start, and Top of the Input List, thus removing the current card image from the list. Then, using the Top of the Input List, and the Input Offset (INPOFSET--see Next Input Character, compute a new Input Character Count (INPCHCT) so that it points to column 84 of the card we just released. Also zero Current Character (CURRCHAR). This means that the next time Next Input Character is asked for a character, it will get the one in column 1 of the next card. NOCONTST (No Continuation Test) also relies on these values being set that way.

2.3.6 Print If Summary PRIFSUM)

When no source listing is being printed, we nonetheless want to print out certain of the source lines as an aid to the user in identifying the program. For example, if a number of programs are compiled without source or object listings, each will have a summary, but it will be hard to decide which summary belongs with which program. So we print out the first line, last line, and all entry point lines (SUBROUTINE, FUNCTION or ENTRY). However, if the user has specifically requested No Summary, then he obviously wants as little output as possible, so we do not print out these lines. In addition, if the source input is coming from the typewriter, there is not much point in printing these lines out, since they were just typed in on the sam typewriter that they would go out on.

So, each of the lines mentioned above executes a call on PRIFSUM, which sets the PRINTRIG unless NS has been specified or input is from the typewriter. (The latter is determined by "Test for SI from Typewriter", Section 4.4.4.4.) The PRINTRIG is tested by "Send Print Buffer", above, to decide whether to create and print a source line even though LS was not requested.

2.4 Character Translate Table CHTRATBL

The Character Translate Table originated in the 900 Series FORTRAN II. In that compiler, it actually translated characters; in ours it does not, but the name has remained for consistency. (The reason that it translatee characters was that some input characters wanted to be treated as if they were other characters. For example, the brackets were treated as parentheses, the space (12) was treated as a blank (60), and this was done by not using the character value directly, but using that value to access a table that told you what the proper character value was.) The Character Translate Table in this compiler serves three purposes:

1. Bits 16-31 (the right-hand half word) contain flags used by the SOC (Set on Character Class) POP. There are a number of "classes" of characters. Each has a bit position. If a character is in that class, the corresponding bit is set. For example, all the letters have the LETRFLAG bit set, so that in one instruction (SOC LETRFLAG) the compiler can find out whether a character it is scanning is a letter or not. The STALFLAG is attached to those letters that can begin a FORTRAN statement (not counting assignment or statement function statements). For example, there are statements beginning with P (e.g. PRINT) but none beginning with X.

   The SOC POP works like a logical OR. If the flag it wants to test for has several bits set, then it answers true if any of those bits are present in the letter in question. For example, the LETDIGFG has both the letter and digit bits set. nnnnnnnnnnnnnnnnTherefore all digits and all letters will test true. 2. Bits 8-11 and 12-15 are needed only because of the IMPLICIT statement. Bits 8-11 contain the implicit type that is to bbe associated with each letter. This is initialized by a routine called Initialize Character Translate Table (INICTTML), which is called during Compiler Initialize in Pass 1. It is initialized such that the letters IJKLMN have integer type and all the others have Real, unless the ADP option was requested, in which case all the others have Double Precision. Whenever an

   IMPLICIT statement needs to set certain letters to a particular type, it changes this field. Note that the position of this field is the same as the mode field in pointers, so that, theoretically, the SOM POP will work on it.

   Bits 12-15 (actually only 12 and 13) contain special flags used by the IMPLCIT statement (and RETTYPE) to indicate whether the type that resides in the mode field has actively been declared or used, or whether it is free to be changed. (See IMPLICIT Statement, 3.5.6).

3. Bits 0-7 contain the character value,. In this compiler, the character value is always the same as the position within the table, so this is redundant and not very necessary, but when we save characters on the Alpha List, for example, (see Saving Alphanumeric Characters, 2.5) we need a full word character value, so we might as well have it sitting here ready to be used, rather than having to create one each time. In addition, this

table could be used in the future to actually do character translation, if there turned out to be a need for this.

On earlier compilers, the Character Translate Table consisted of fixed data. With the advent of the IMPLICIT statement on Sigam, however, the table is now stored into; therefore, it has to reside in non-protected storage. Most of it (bits 0-7 and 16-31) is still initialized by loading and is never changed. The other bits are initialized actively and may be changed.

2.5 Saving Alphanumeric Characters

There are two places in the compiler where we have to accumulate alphanumeric characters into a text string, but at the same time, we want to do syntax scanning on the characters. This differs from ordinary alphanumeric scanning in specifically defined alphanumeric fields. For example, a constant such as 4HABCD or 'GOBBLEDYGOOK' contains alphanumeric characters that need to be accumulated, but we can just step through and accumulate them; we do not have to do anything else special with them. However, in a statement such as

```
FORMAT (3X,F9.3/14)
```

the whole text between the parentheses needs to be saved up and built into a text string, but we also have to do sy ntactical scanning on the format specifications. This is handled by the following steps:

1. Set the Save Active Characters Flag (SVACTFLG) so that the Next Active Character routine (NXTACTCH) will collect all the active characters on the Alpha List (ALPHALST). Note that blanks are not considered active except within Hollerith fields (and in-line assembly code statements) which is just fine for FORMAT statements too.

2. Proceed with the statement, making no special considerations for the fact that Next Active Character is doing something special down below. That all happens automatically.

3. After scanning the final right parenthesis, call Pack Alpha List (PKALPHML), which takes the characters off the Alpha List in forward order (i.e. starting at the Top) and packs them, four characters per word, and files them on the Code List, preceded by a word count. The characters are put onto the Alpha List (by NXTACTCH) in their full-word Character Translate Table form (see Character Translate Table). In this form, the actual byte containing the EBCDIC character is the leftmost byte.

Note that this use of the Alpha List is one of the few places where information is put onto the bottom of a list and then taken off the top. This has the effect that the list moves through memory, disrupting the status of the dynamic list allocation. We call this a Freight Train (see 1.4.10), and it is prevented by reserving the Alpha List before we begin, so that the Top is held firm, and releasing it when we are through.

The other statement that uses the Alpha List is OUTPUT, e.g.

```
OUTPUT X , A+B , (C+D)/E ,  (P(K), K=1,10
```

Here the procedure is essentially the same, but more complicated, because we may have to create several separate text strings (i.e. several FORMATs), packing up one and then beginning another. Furthermore, whereas the scanning of FORMAT statements is all done right there within the section that handles FORMAT statements, the OUTPUT statement has to handle expressions, DO-loops, etc. and goes off to outside routines to do that. Meanwhile, it turns on and

off the SVACTFLG so as to save characters at just the right places. For example, the left parenthesis in list item three above is part of the expression, and needs to be saved, while the left parenthesis in the fourth list item is part of the implied DO-loop and not the inner expression, and therefore should not be saved.

2.6 Diagnostics

There are two ways in which diagnostic error messages are produced. Most of them occur during the scanning of a statement and are associated with a particular character in the statement. As described below, these messages are not printed at the time the errors are encountered. They are saved up and all printed when the statement is completely scanned. These nare called delayed error messages. Other messages are not detected during scanning and can thus be printed immediately. These are called immediate error messages.

2.6.1 Format of Error Messages

All compiler error messages are printed in one of the following two formats:

```
WARNING: message...

ERROR: message...
```

WARNING indicates that a recovery has been made. ERROR indicates that the statement has been deleted. The severity of an error is contained within the message itself; it is not determined by the routine that puts out the diagnostic. Each error message begins with a one-byte key that determines its severity. This is followed by the text of the message, enclosed within dollar signs. (See also Quotes 1.15, and Print POPs 1.14.) Most of the error messages are contained within STASCAN and are produced by the ERR Proc. For example,

```
ERR,2          'NUMBER OUT OF RANGE.'
```

has the effect of:

```
GEN,8,8,16     X'02 ,'$'.'NU'
TEXT       'MBER'
TEXT       'OUT'
TEXT       'OF'
TEXT       'RANG'
TEXT       'E.$ '
```

If the severity is greater than two, the sign bit (first bit) of the key is also set, as an easy-to-use indication that the statement must be deleted. Note that the interval severity used by the compiler is a holdOover from the 9300, and is not the same as the external severity used to interface with the rest of the system. The correspondance is shown in the following table.

Internal Severity	Error Key	External Severity	Meaning
1	X'01'	4	Minor error, good recovery (WARNING)
2	X'02'	7	Major error, some recovery (WARNING)
3	X'83'	10	Major error, no recovery, statement dele
4	X'84'	15	Abort error, no recovery, compilation ab

This is also discussed in the section on the Summary Listing, 2.30.

There are some error messages in MERRIMAC (the Monitor Interface routine) that do not follow this format exactly. They probably ought to, but it is not too important, since most of them abort immediately anyway, so their severity is a moot point.

2.6.2 Immediate Error Messages All immediate error messages should begin by EATing the location of the desired error message and calling PRERR1 (Initialize to Print Error Message). This routine looks at the error key, prints either "WARNING: " or "ERROR: ", increments the Number of Error Messages (NERRMSGS) by 1, and updates the Maximum Error Severity (ERRLEVEL) if this message is of higher severity than any previous ones. These two counts are used by the Summary Listing.

Following a call on PRERR1, the caller must put out the rest of the error message. In PASS1F there is a routine that does both of these operations, called PRERRMSG. However, not everybody can use this, since some error messages are rather complicated and have to be built up, part by part (e.g. allocation errors or missing labels). PRERRMSG is also used by delayed errors (discussed below), when it is finally time to actually print them out.

Immediate error messages are used in the following places: in PASS1F, they are used by missing END cards (see Label Field Scan, 2.9.5), unclosed DO-loops (see Check for Unclosed DO-loops, in 2.9.4),and by undefined labels (see Check for Undefined Labels in 2.9.4). In PASS0F, they are used by the ABORT routine, which in turn is used by various other places when the compilation needs to be aborted (see Abort Errors). In MERRIMAC, they are used by ILPRABRT when aborting (see Abort to Monitor), and upon error while reading a scratch file (see XREAD). In PASS2F, they are used by P2ABORT before calling ABORT (in PASS0F), and by the Asinine Return Statement Gen (when an IBM RETURN asterisk count is too high). In ALLOCATE, there is a single place through which all allocation errors are funneled. These are all immediate messages. And in SUMMARY, the message about total program size too large is immediate (see Output Final Statistics 2.30.6). All other error messages are delayed.

2.6.3 Delayed Error Messages

There are two reasons why errors detected during scanning are not printed immediately, but must be delayed.

1. If there are multiple errors on a statement, they have to be printed in forward order (by character position), which is not necessarily the order in which they are discovered. For example:

```
FORMAT(F10.3

WARNING: STATEMENT HAS NO LABEL AND CANNOT BE ACCESSED.
WARNING: MISSING RIGHT PARENTHESIS ASSUMED HERE.
```

   It is not until we know that this is a FORMAT statement that we know it must have a label.

2. More important, an error message, when detected, may not be the least bit pertinent or valid, and may never actually come out. Because of our "TRY-Fail" method of scanning, errors may be detected while scanning for one kind of statement, and then discarded when it turns out that this is actually a different kind of statement. For example, the second statement below will be scanned first as a DO statement (because it begins with "DO"), and will produce two error messages:

```
3 COMPLEX DO3K(10,10) , C
  DO3K = C

WARNING: LABEL HAS PREVIOUSLY APPEARED.
ERROR: ILLEGAL TYPE.
```

   It will then be rescanned as an assignment statement, producing an entirely different error message:

```
DO3K = C

WARNING: ARRAY USED WITHOUT SUBSCRIPTS; FIRST ELEMENT ASSUMED.
```

   The first set of error messages are not pertinent and are never used.

All delayed error messages are accumulated on the Error List athe time they are discovered. Each error produces a two-word entry on the Error List:

1. Character count of the character under which the error marker is to be printed.

2. Location of the error message that is to be printed (key and text).

There are three POPs used in setting up these entries:

ERC ERror on Current character

ERC puts the Input Character Count (INPCHCT) on the Error List, unless Current Character (CURRCHAR) is an ENDCHAR, in which case it takes the Last Active Character Count (LACTCHCT), adds 1 and uses this. This is so that a statement such as

```
X = A +
```

will produce the error marker as shown instead of over in column 73.

ERL ERror on Last active character

ERL puts the Last Active Character Count on the Error List. This places the error marker under the last non-blank (usually) character that was scanned before the current one.

ERW ERror on character count specified in WO

ERW is used to put an error marker on a character scanned at some previous time. The character may be nowhere near the current character (but it must be in the same statement). At any time during scanning, the compiler may save away the Input Character Count, then later pick up that value into WO and put an error message there using ERW. ERW also removes the count from WO.

These three POPs all do one further thing of importance. As discussed under TRY-Fail, there are various wasy in which a scan (either at the statement level or under a TRY) can fail. Some of them are drastic and unrecoverable and go directly to FAIL. Often, however, an error is detected which is fatal (we cannot generate code for it), but it is possible to continue the scan in the hopes of detecting further errors. Therefore, when one of the above POPs is executed, it looks at the key of the error message to determine whether the error is a fail error or not (i.e. whether the statement will later have to be deleted). If the sign bit of the key is set (i.e. the severity is greater than two), it sets the sign bit of the delayed fail flag, which is kept on the TRY List. (Since delayed fails can occur at various recursive levels, the delayed fail flag must be recursive. This is described under TRY-Fail.) In addition, if source input is on-line (see SIONLINE, 4.4.4.4), then <u>all</u> errors are considered fatal. Since the user is presently typing in his program, we can give him a chance to correct any statement that has even a minor error in it. (Unfortunately, we give him no way to <u>force</u> us to accept the statement even though it has an error, and there may be cases in which he would want to do this. For example, the compiler is capable of calling a function with no arguments, but it insists on giving a diagnostic. This means that on-line, there is no way to call a function with no arguments, whereas in batch mode there is.) If we are on-line, we unconditionally set the delayed fail flag. Later, either at Statement Exit or when TRY thinks it has succeeded, this flag is tested to determine wheteher to fail at that time (see TRY-Fail).

Note that the error POPs do not update either NERRMSGS or ERRLEVEL, since the errors they are producing may not apply. This is done by PRERR1, which is called (via PRERRMSG) by Release Card (RELCARD), when the statement has been satisfactorily scanned and is being printed out (see Card and Line Handling, 2.3).

2.7 TRY-Fail

The TRY and Fail procedures may well be the most compilcated section of the compiler. Also, in their evolution from the 900 Series FORTRAN II, they have become rather clumsy and inefficient, as will be shown below. The purpose of these procedures is to allow multiple scans of statements or parts of statements. This is necessary because FORTRAN is an ambiguous language. When you start to scan something, you do not necessarily know what it really is, and the information gathered may be erroneous. Consider, for example, the statement

```
DO3K = I
```

Because it begins with "DO", we assume this is a DO statement. As we scan, the number 3 gets registered as a statement number, K gets registered as an integer control variable, and I gets registered both as an integer scalar and as the initial DO-control expression. Then the statement fails, for lack of a comma and another DO-control expression. When we rescan it as an assignment statement, we find that statement number 3 does not exist at all, nor is there a variable K, nor is I an initial DO-control expression. All there is is a variable DO3K set equal to a variable I. In order to back-track this way, it is necessary to be able to back up the scan (which involves three counters: Input Character Count, Last Active Character Count, and Current Character -- see Character Scanning) and also to throw away information that has been gathered during a scan.

2.7.1 Saving and Restoring Information

There are three techniques used for this. The first two involve the SAVELIST and the third involves the SYTCHLST (Symbol Table Change List). Some of the things that need to be saved before trying a scan that may fail are lists and others are individual words. For the most part (the SYTCHLST handles the exceptions), the only way in which a list will change during a scan is by having words added to it. Thus, to get back the way it was before, it is only necessary to remember where the Start, Top, and Bottom were relative to the Base (see Dynamic Lists). Furthermore, most of the time, the Start and Top of a list are the same as the Base, which means only having to remember where the Bottom was. Thus "saving a list" does not mean putting all its information somewhere; it only means remembering how big the list was, so that the extraneous information added later can be discarded. List parameters are saved on the SAVELIST by the SAL POP.

SAL Save A List

SAL puts from one to four items on the SAVELIST. Whichever of the values in the first column below are non-zero are moved onto the SAVELIST

Bottom-Top (00F00000)

Top-Start (0F000000)

Start-Base (F0000000)

Finally, the bottom word on the SAVELIST is the value of the list number. This identifies which list is being saved. Into the upper 12 bits of the list number word are merged flags to indicate which of the above three values were actually saved. The fllags are shown in the second column above. For example, if the first hex digit is an F, this means that the Start and Base are different (i.e. the list has been reserved), and the amount they differed by is contained in the next word (moving from Bottom to Top) on the SAVELIST. If that digit is not an F, then that value is not present and should be assumed zero. In addition, the list value word always has the sign bit merged into it. This indicates that this is a saved list, not a saved value (see below). The following three-word entry might occur when saving a list that had not been reserved, but had had two words taken off the Top:

00000005 (Bottom-Top)

00000002 (Top-Start)

8FF00027 (List number X;27')

Individual values (such as the Input Character Count(are also saved on the SAVELIST using a two-word entry. The first word (towards the Top) is the item's value, and the second word (on the Bottom) is its location. Notice that the "location" word odes not have the sign bit set, which indicates that this is not a saved list entry.

Restoring of saved lists and values id done by RESTORML, which is called by FAIL, as we shall see below.

The above method of saving lists does not work when they are being changed not by having information added on to the bottom, but by having values changed that are already on the list. This happens primarily to the Symbol Table (the REG and SUP POPs go back and change old entries in the Symbol Table), but can also happen to the two label tables (because of the SUL POP). These lists, in addition to being saved in the normal way (described above), are protected by the SYTCHLST (Symbol Table Change List). This is a list of two-word entries, set up any time an already existing word in a list is changed. The first wordis a pointer to the proper location in the proper list (see Pointers) and the second word (on the Bottom is the old value. STYCHLST entries are also restored within the FAIL routine.

2.7.2 Overall Statement Scan and Fail Structure

Before discussing who saves what and when and why, it is necessary to look at the general layout of how statements are scanned. There are three mutually ambiguous classes of statements: assignment statements, statement function definitions, and statements that begin with a key word, or quote (which we call quoteOtype statements, and which constitute everything that does not fall into the first two classes). For example, consider the statement

```
DO3K(I) = I
```

At first it looks like a DO statement (quote-type), but fails at the left parenthesis. It can be an assignment statement, but not if DO3K is not an array, in which case it may be a statement function definition. We try the three classes of statement in that order: quote-type first because if the statement actually begins with one of our key quotes, the chances are very good that that is what it is. Statement functions come after assignment statements because they are much less common. Essentially the procedure is to save certain data and try scanning for a quote-type statement. If that fails, restor the data and try scanning for assignment statement. (Actually, if we do not even find an initial quote, none of the data has been changed, so we do not even find an initial quote, none of the data has been changed, so we do not have to restore anything.) If assignment statement fails, restore the same data again and try statement function. If that fails, this is aan illegal statement, and we have to delete it and put out diagnostics. The deleting is easy; restoring the data again puts us back where we were when we started. Putting out the diagnostics is less obvious. Each of the three calsses of statement has been accumulating diagnostics, based on the assumption that what it is seeing is indeed that kind of statement. (There is an example of this in the section on Delayed Error Messages, 2.6.3.) We want to select the calss of statement that seems most likely to be what the user had in mind, and we make that decision based on which one got further through the statement before it gave up. As each class of statement fails, a routine called FEXFAIL keeps track of which one got the furthest and hangs on to that set of diagnostics. Naturally if any class succeeds (even with non-fatal error messages), then it does not matter how far previous classes got; those old error messages are thrown away and the new ones (if any) are used.

The above procedure is complicated a good bit by several factors. One is that it has to work recursively because of the Logical IF statement. While that process is in operation for the scan of what looks like a Logical IF statement, the Logical IF statement has to call Statement Scan to scan the statement (or statements) that it controls, and the same multiple scan has to go on with <u>that</u> statement.

Another factor that requires the fail procedure to be recursive is the TRY POP, which will be described later. Essentially it makes it possible to do multiple scans on parts of statements (usually parts of expressions), without abandoning the whole statement.

A third complicating factor was the introduction compound statements into the language, and a fourth is the fact that even certain of the quoteOtype statements are ambiguous with one another. For example, the statement

```
READ TAPE, LIST
```

starts out looking like a READ TAPE statement, but it turns out that it is not; it is a READ statement addressing the array TAPE, which contains the FORMAT (or the scalar variable TAPE which has been ASSIGNed). This means that we have to do multiple scans within certain of the quote-type statements.

A few further pieces of information are needed before you try to make your way through the "flowcharts" that follow. First, be sure you have read the sections on Character Scanning, 2.2, (so that you understand the use of the various counts) and Diagnostics 2.6, (so that you understand delayed error messages).

The FEX POP (Set up Fail Exit) is used to indicate what class of statement to scan next if the current scan fails. It is a simple-minded POP, and probably should not be a POP at all, since it is used so seldom. All it does is store its effective address in the cell FEXEXIT. When a scan fails, it goes to FAIL, which gets to FEXFAIL because of the contents of the cell FAILEXIT which is set up elsewhere (under a TRY, FAIL goes to TRYFAIL), and FEXFAIL gets to the next scan class by using the address in FEXEXIT (whcih was set up by the FEX POP).

The TRYLIST is used both at the statement level and by the TRY POP (which will be described later). At the statement level, it contains a four-word entry. (If we recurse, because of Logical IF, there will be a new fourOword entry. If we recurse because of a TRY, it uses only a two word entry; the first two words are not needed.) These four words are:

1. OLDCHCT (Old Caracter Count). Used by FEXFAIL in determining whether a new scan got further than the old one.

2. FEXEXIT. Where FEXFAIL is supposed to exit to. This has to be recursive because, under a Logical IF, FEXFAIL will be trying other scans on the controlled statements.

3. The Delayed Fail Flag. This has neither a symbol name nor a fixed location. It is always and only carried in this, the next to the bottom word of the TRYLIST. It is set by any error POP that registers a failOlevel error on the Error List (i.e. internal level 3). Later just before Statement Exit, this flag is tested to see if the statement really did succeed. (Under a TRY, this is tested within the TRY POP itself.)

4. The Automatic Succeed Flag. Again, this is carried only in this, the bottom word of the TRYLIST. It is used only by the TRY POP (described later), and indicates that a TRY must act as though it succeeded even though it failed.. Actually it is not quite true that only TRY uses this. Logical Expression Scan will set this flag even when it has not been called by a TRY; it will never be tested, but it must be there. The other reason it must be thre, even if it is not used, is so that the Delayed Fail Flag *which is used) will be the next to the bottom item on the TRYLIST.

The following paragraphs contain flowchart-like, step by step descriptions of the statement scan and fail procedures, extracting only those operations that pertain to fail recovery. They are not actually flow charts because many of the steps require too much explanation of why they are doing what they are doing to ever fit in boxes.

A. Overall Statement Control

A.1 Save (FAILEXIT=LINEFAIL). This means save, as an individual value on the SAVELIST, the value of FAILEXIT as if it now contained the address of LINEFAIL. In other words, if there is a fail at the line elevel (which is above the statement level), FAIL will restore FAILEXIT to contain LINEFAIL, thinking that that was the value it had when it was saved. Actually, we do not really give it that value here, because it will not be needed unless there is a fail, in which case it will properly be "restored" to that value.

A line level fail is needed both in case the label field fails and because there may be several statement level fails without meaning that the whole line fails (which causes, for example, the label to be thrown out).

A.2 DELTRIG= FALSE. This indicates that the statement has not been deleted. If necessary, it will be changed by ILLEGSTA (paragraph I).

A.3 Save the lists that might be added to by Label Field Scan, namely the WORKLIST, Local Label List, Non-local Label List, Symbol Table (only because of F in column 1, probably), CODELIST, and EXITLIST. Note that most of these are saved down in the routine LINEINIT, but the EXITLIST HAS TO BE SAVED AT THE LEVEL we want it restored to. That is LINEINIT cannot save the EXITLIST, because it is just about to return to the upper level, thus removing one exit from the list. We want the EXITLIST saved without the return from LINEINIT on it.

A.4 Go off to Label Field Scan (described in 2.9.5 elsewhere). If it returns (i.e. succeeds), jump to step A.8.

A.5 (LINEFAIL). Set CURRCHAR equal to minus one so that we definitely will not find a semi-colon in step A.12. If the label field fails (e.g. because of a semi-colon in column 4), we do not want the overall control thinking that it can nonetheless go forth and scan a compound statement because it sees a semi-colon as the next character.

A.7 DELTRIG = TRUE. The statement will be delted. This is tested by Release Line (see 2.3.1).

Now jump to step A.11 (LINEFIN).

A.8 Label Field Scan has succeeded. It returns with an answer, which is <u>false</u> only if it found an F in column 1. In this case, it has scanned the

whole statement for us, and we do not have to call Statement Scan, so jump to step A.11 (LINEFIN) immediately.

A.9 Otherwise, JRS to STASCAN (Statement Scan, paragraph B). It may have several fails at its level, but if it is finally unable to find a legal statement, it will return with answer false, in which case we will go directly to FAIL, in order to fail at the line level also. The path in this case is exactly the same as that described beginning in step A.5.

A.10 If STASCAN returns with answer true, proceed to step A.11 (LINEFIN).

A.11 (LINEFIN). Release the SAVELIST and SYTCHLST. We have now either succeeded or failed, but in any case, wer are not going to try anything else on this statement, so we will not have to restore any of this information (some of it may already have been restored by FAIL).

A.12 If the next character is a semi-colon, call COMSTASC (Compound Statement Scan, paragraph J). This will happen only if neither the label field or the first statement failed (because of step A.6.). COMSTAC will return with an answer whether any of its statements failed. We can ignore the answer; if they failed they failed, and no code was generated for them. We do not have to know about it. That answer is used only by Logical IF, which wants to fail if any of the statements it controls fails.

In any case, proceed to step A.13.

A.13 If this was the END statement (as indicated by NOTENDTR being false), that is all, go to FINISHUP.

Otherwise, return to step A.1 (NEXTSTA).

B. Statement Scan STASCAN)

B.1 Put OLDCHCT and FEXEXIT onto the TRYLIST. (This is explained just before paragraph A.) I think this is necessary only if STASCAN actually is curren ly being called recursively by the Logical IF statement, but lower level routine are not supposed to know about such things, so we do it unconditionally; it doesn't cost much. Tehe values will be restored in STAFINAL (step H.3).

B.2 Save (FAILEXIT-FEXFAIL). (See step A.1). If a scan faisl, this will cause FAIL to go to FEXFAIL (rather than TRYFAIL).

B.3 Call FEXTRYSV (FEX-TRY Save, paragraph C). This does most of the preparation for being able to fail at the statement level or under a TRY.

B.4 Save a variety of lists. This means any important list that any statement might change, which is most of them. Some of these are saved in a subroutine called EXPSAVE, which is also used by various TRYs, and saves the lists that might be changed by scanning an expression. The other lists are those which would not be changed by expressions, but might by other statements.

B.5 OLDCHCT = -1. Initially no class of statement has gotten any distance through the statement. This will be updated by step F.2.

B.6 Reserve the SVERRLST (Save Error List), in case of recursion. This list will be used by FEXFAIL (paragraph F) to save the = error messages of the scan that got the furthest. Note that FEXTRYSV reserves the ERRORLST but does not reserve the SVERRLST because TRY does not need it. The SVERRLST will be released by STATEXIT (step G.1) if any statement succeeds, or by ILLEGSTA (step I.4) if none does.

B.7 FEXEXIT = ASIGNMNT. This is done with the FEX POP and tells FEXFAIL to go try Assignment Statement if the

quote-type statement scan fails.

B.8 Go off and scan for a quote-type statement (see Scan, paragraph D). If it succeeds, it will go to STATEXIT (paragraph G), which will exit TRUE from STASCAN.

B.9 If it fails (for any of a variety of reasons described in paragraph D), it will go to FAIL (paragraph E), which will go to FEXFAIL (paragraph F) because of step B.2, and FEXFAIL will go to ASIGNMNT (step B.10, below) because of step B.7.

B.10 (ASIGNMNT) FEXEXIT = ASFD. Like step B.7, but tells FEXFAIL to go to scan Statement Function Definition next.

B.11 Go off and scan for an assignment statement (see Scan, paragraph D). If it succeeds, it will go to STATEXIT (paragraph G), which will exit TRUE from STASCAN. (Note that steps B.10-12 and steps B.13-15 are essentially the same as steps B.7-9.)

B.12 If assignment statement fails, it will go to FAIL (paragraph E), which will go to FEXFAIL (paragraph F) because of step B.2, and FEXFAIL will go to ASFD (step B.13, below) because of step B.10. Note that FAIL has now restored FAILEXIT to point to FEXFAIL twice. The information on the SAVELIST is used to restore values and lists, but it is also left on the SAVELIST in case it is needed again. It is not released until STAFINAL (see step H.1).

B.13 (ASFD). FEXEXIT = ILLEGSTA. This is the last chance. If statement function definition fails, FEXFAIL will go to ILLEGSTA (paragraphI).

B.14 Go off and scan for a Statement Function Definition (see Scan, paragraph D). If it succeeds, it will go to FEXFAIL (paragraph F) because of step B.2, and FEXFAIL will go to ILLEGSTA (paragraph I) because of step B.13. ILLEGSTA will clean up and then exit FALSE from STASCAN.

C. FEX-TRY Save FEXTRYSV)

C.1 Reserve the SAVELIST, so that if a fail occurs, only the information saved at this recursive level will be restored. The restoring is done by FAIL (see step E.4), but without taking it off the SAVELIST. The information will be released in STAFINAL (step H.1) when doing FEX-type scans, or within the TRY POP (step K.10) or TRYFHIL (step M.1) when doing a TRY.

C.2 Save (as values on the SAVELIST) the Input Character Count, Current Character, and Last Active Character Count. These will also be restored by FAIL (step E.4), thus restoring the scan pointers to exactly where they are now.

C.3 Move two values of FALSE onto the TRYLIST. These are the initial values of the Delayed Fail Flag and the Automatic Succeed Flag. The Automatic Succeed Flag is used only by TRY (see steps L.2 and K.6, and M.3). When under FEX, the Delayed Fail Flag will be set if a fail-level error occurs (step D.1); it will be tested just before STATEXIT (step D.4); and it will be removed by STAFINAL (step H.2). Under TRY, the Delayed Fail Flag will likewise be set if a fail-level error occurs (step L.1); it will be tested within the TRY POP (step K.5); and it will be removed within the TRYPOP (step K.8).

C.4 Save the WORKLIST and ALPHALST. These are the only lists that always need saving by both FEX and TRY. Under FEX, other lists of course get saved; this is done in STASCAN (step B.4). Under TRY, each TRYed routine must save whatever lists it thinks it might add to. The ALPHALST needs to be saved because of the OUTPUT statement only. When that statement is in operation, expressions and other things are scanned and Next Active Character is busy putting away characters on the ALPHALST and the expression scanners do not realize this is happening. So we have to make sure that the ALPHALST gets restored if a scan fails.

C.5 Reserve the Error List. As we do any particular scan, we may pile up errors on this list, and we have to be able to deal with them later (throw them out or move them to the SVERRLST or whatever) without disturbing errors at previous recursive levels. Under FEX, these errors may get copied over to the SVERRLST by FEXFAIL (step F.3); the Error List will be released if no statement succeeds (se step I.1); the Error List will be unreserved when a statement succeeds (step G.2). Under TRY, the Error List will either be released by TRYFAIL (step M.6) or within the TRY POP itself (step K.9).

C.6 Reserve the SYTCHLST. As with the SAVELIST (in step C.1), this information will be restored by FAIL (step E.3). Under FEX, the SYTCHLST will be released by STAFINAL (step H.1). Under TRY, the SYTCHLST will either b released by TRUFAIL (step M.1) or will be unreserved within the TRY POP (step K.9).

C.7 Exit.

D. Scan

(This contains the important structural parts of each of the scans for the various classes of statements (quote-type assignment, statement function). It is called in steps B.8, B.11, and B.14.)

D.1 If, at any time during the scan, an error of severity 3 occurs, the Delayed Fail Flag (on the TRYLIST, see step C.3) will be set. This is done by the error POP (ERC, ERL, or ERW).

D.2 If, in addition, the scanning routine considers it impossible or undesirable to try to continue the scan any farther, it will go directly to FAIL (paragraph E). FAIL will go to FEXFAIL (paragraph E), which will go wherever previously directed by STASCAN (paragraph B).

D.3 Otherwise, control will end up at one of the statement exits (e.g. ACTIVEX, JUMPEXIT, DECqEXIT). If any reason is found that this otherwise successful statement should not be allowed (e.g. not allowed under Logical IF), control will then go to FAIL, with the same results as in step D.2 (except that we scanned the whole statement, thus possibly uncovering further error messages).

D.4 Finally, before allowing the statement to succeed, we have to check the Delayed Fail Flag, which may have been set in step D.1. If it was, we again have a reason to go directly to FAIL.

D.5 If the statement truly succeeds, we go to STATEXIT (paragraph G), which will exit TRUE from STASCAN.

E. FAIL

E.1 Make sure Current Character is not empty. (This is done with a CSK for anything.) This is because later on (in ILLEGSTA, step I.2) we may want to get back to our current scan point. In order to determine the scan point exactly, however, you need to know both the Input Character Count (INPCHCT) and the contents of Current Character (CURRCHAR). If CURRCHAR is empty, INPCHCT really points to the next character, and will do so as soon as a character is needed. By making sure that CURRCHAR is not empty, we can guarantee ILLEGSTA that it does not have to worry about whether INCPHCT is accurate or not; it will always point to the "next" character.

E.2 Set NEWCHCT = INPCHCT. FEXFAIL (paragraph F) is going to be interested in which of various scans got the furthest through the statement, so it wants to know what the INCPHCT got to. However, we are just about to restore the INPCHCT to what it was before we started this scan (so that we can try another scan), and at that point, the value of INPCHCT will have been lost. So we save it in NEWCHCT.

E.3 Restore all the current SYTCHLSTitems (old values in lists that have been changed by REG, SUP, and SUL POPs). In the process, we empty the SYTCHLST (since we will never have to restore those particular changes again), but do not release it, because there may be changes in a subsequent scan. It is essential to restore these items from Bottom to Top to get the values back to their original values. Suppose a certain list value changed from X to Y and from Y to Z. If we restored them from Top to Bottom, we would restore it to X and then to Y, which would be wrong.

E.4 Call RESTORML to restore all the list parameters to the way they were when last saved (in step B.4 or C.4). This does <u>not</u> empty or release the SAVELIST. If we do subsequent scans, we will have to again restore all this same information. (That only happens under FEX; under TRY, the SAVELIST will soon be released by TRYFAIL (step M.1) or the TRY POP (step K.10).)

Note that certain single values are also restored here, including the INPCHCT, which is why we saved it in NEWCHCT above. Another single value that gets restored is FAILEXIT, which was saved either in STASCAN (step B.2) when under FEX, or in the TRY POP (step K.1) when under TRY. Or it may have been saved at the upper line level (step A.1).

E.5 Go to the address specified by FAILEXIT, which has just been restored. This will either be FEXFAIL (paragraph F) or TRYFIAL (paragraph M) or LINEFAIL (step A.6).

F. FEXFAIL

F.1 If NEWCHCT is not greater than OLDCHCT (i.e. this scan got no further than the last one), jump to step R.4. The reason we demand that a subsequent scan get further (not just as far) is that the three classes of statements are scanned in decreasing order of probability. If a statement function gets as far as an assignment statement, we would rather assume the statement was an assignment.

F.2 If this scan did get further, update OLDCHCT = NEWCHCT.

F.3 And empty the SVERRLST and copy any errors currently on the ERRORLST over to the SVERRLST. Since we have decided that this scan takes precedence over the old one, we no longer need the old error messages.

F.4 In either case, empty the Error List. Either we do not need these new error messages, or we have saved them on the SVERRLST.

F.5 Reset the Delayed Fail Flag (on the TRYLIST) to false. It may have cuase a fail on this last scan, but now that a new scan is going to be attemtped, it no longer applies.

F.6 Turn off the DELTRIG (indicating statement deleted). There ia a very subtle reason why this is necessary, and it points up some of the unpleasant complications caused by the ambiguity of the FORTRAN language. Suppose you try to write a statement function named IF that takes one Logical type argument and returns a Logical result. For example:

```
LOGICAL IF, L, FLAG

IF (L) = L .OR. FLAG
```

That statement function is perfectly legal, but it causes problems. The statement first gets scanned as a Logical IF statement, and it gets fairly far. It finds "IF" and a Logical expression (L). Then it goes off to look for a statement to control and it immediately fails on the on the equal sign. This takes it to ILLEGSTA (paragraph I), which sets the DELTRIG, thinking that the statement will be deleted. Indeed it would, if this were in fact a Logical IF, but it is not and the DELTRIG is not maintained recursively. This is because we had not thought of this example originally, and since it appears to be the only example where this happens, it is easier to fix it here -- just in case -- than to change it to work recursively.

F.7 Turn off the SVACTFLG (Save Active Characters Flag). This may have gotten turned on while scanning what we thought was a FORMAT or OUPUT statement, which failed before it had a chance to turn the flag back off.

F.8 Go to the address specified by FEXEXIT. This will either be ASIGNMNT (set up in step B.7) or ASFD (set up in step B.10) or ILLEGSTA (set up in step B.13).

G. Statement Exit STATEXIT)

(Control comes here from STASCAN (paragraph, via paragraph D) when one of the classes of statement scan succeeds.)

G.1 Release the SVERRLST. Since this statement succeeded, we are not interested in any error messages that were collected and saved by previous statements that did not succeed.

G.2 Unreserve the Error List. We are, however, interested in any error messages collected by this successful scan, so we will combine them with the upper level (which is usually line level, except under a Logical IF).

G.3 Fall into STAFINAL (which is the part of STATEXIT that is also needed by ILLEGSTA). STAFINAL will exit TRUE from STASCAN.

H. Statement Final STAFINAL)

H.1 Release the SAVELIST and the SYTCHLST. These were reserved by FEXTRYSV (paragraph C), and FAIL (paragraph E) has been playing around with them, but now we are all done with this statement, and they are no longer needed. There may currently be unrestored information on both of these lists (or there may not), but it does not have to be restored; we now know that whatever changes were made to lists are correct and will stand.

H.2 Remove the Automatic Succeed Flag and the Delayed Fail Flag from the TRULIST. (They were put there by FEXTRYSV, paragraph C.)

H.3 Restore the old values of FEXEXIT and OLDCHCT that were saved on the TRYLIST by STASCAN (step B.1). This has an effect only if the statement we just finished scanning was under a Logical IF. In that case, the FEXes and OLDCHCT comparisons we were doing on the controlled statement are irrelevant to the ones in force at the Logical IF's level, and we have to go back and start using those again.

H.4 Exit TRUE. On a successful statement (we got here from STATEXIT), this will exit TRUE from STASCAN. On an illegal statement (we got here from ILLEGSTA), we will go back to ILLEGSTA, which will ignore this ture answer.

I. Illegal Statement ILLEGSTA)

(Control gets here from STASCAN (step B.15), when no class of statement can be successfuly scanned.)

I.1 Release the Error List. We will want to use the error messages of the statement that got the furthest, and these have been saved on the SVERRLST by FEXFAIL (step F.3).

I.2 Set INPCHCT = OLDCHCT-1 and empty CURRCHAR. OLDCHCT tells how far we got on the best scan we did. We want to move INPCHCT out to this point, so it will look like we just scanned out to that point. (Actually, INPCHCT has been restored to the beginning of this statement in prepearation for another scan, but there are no more classes left to try.) There are two reasons for this. First, if we happen to be under a Logical IF, it would like to know how far we got because, in a sense, that is how far it got and it uses this figure in competition at its own level to determine whether it really is a Logical IF. (Just how often this ever has any effect is questionable.) Second, if we are in the middle of a compound statement, it may be possible to go and scan the next part, even though the one we just scanned failed. If we move INPCHCT out to however far we got, and that turns out to be a semi-colon, this may work.

There is one trickiness about moving INPCHCT out. Remember in FAIL (step E.1), we made sure that CURRCHAR was not empty? This means that the value of INPCHCT we are about to set up is valid only if the character it points to is in CURRCHAR, which it is not. However, if we back INPCHCT up by one and empty CURRCHAR, then we are all set to go fetch the next character when we need it.

I.3 Call STAFINAL (paragraph H) to do the standard end-of-statement cleanup. Ignore the answer TRUE that it returns. (That is used when STAFINAL is branched to from STATEXIT (on successful statements).

I.4 Copy the SVERRLST to the ERRORLST and release the SVERRLST. As explained in step I.1, the errors we want are those that have been saved by the statement that got the furthest through the scan. We put these on the Error List (just as a successful statement would have), and we no longer need the SVERRLST.

I.5 Set the DELTRIG (Statement Deleted Trigger) to TRUE. This will cause Release Line to print STATEMENT DELETED (see Card and Line Handling), unless this decision is premature, in which case DELTRIG will be reset by FEXFAIL (see step F.6).

I.6 Exit FALSE from STASCAN.

J. Compound Statement Scan (COMSTASC)

This is called either at the line level (step A.12) or by Logical IF. It will scan as many statements as it can, separated by semi-colons, and return an answer whether all of them succeeded. At the line level, we are not interested in that answer, but Logical IF is, since if any of its statements fail, it must fail too.

J.1 Set a flag (in WD) to TRUE, meaning no statements have failed yet.

j.2 Call Statement Scan (STASCAN, paragraph B), which will return an answer whether the statement succeeded.

J.3 If it did not succeed, set the flag (in WD) to FALSE.

J.4 In either case, look for a semi-colon (see also step I.2). If there is one, return to step J.2.

J.5 If there is no semi-colon, test the flag (in WD). If it is false, exit FALSE, indicating that not all the statements succeeded.

J.6 If the flag is TRUE, exit TRUE, indicating that all the statements did succeed.

Top-Down Programming

Before describing the TRY POP, we need some explanation of the concept of "top-down" programming, as opposed to "bottom-up" programming. This will help in understanding why the TRY POP is needed and why it has become so inefficient.

Top-down programming is analagous to learning about a company by looking at its organizational chart from the top down. First the broadest areas of responsibility are revealed. Within each area, smaller and smaller areas are detailed as you go down the chart, until you get to the people who do the "real work". The syntax of the FORTRAN language can be broken down in much the same way, especially in a simple FORTRAN such as the 900-series FORTRAN II. Consider the IF statement in that FORTRAN. It can be essentially defined as follows:

IF Statement - "IF(" expression ")" label "," label "," label

And to a great extent, the coding can be done that way.

1. Quote scan for the characters "IF(". (Fail if not there).
2. Call expression scan routine.
3. Character scan for right parenthesis.
4. Call label scan routine
5. Character scan for comma.
6. Call label scan routine.
7. Character scan for comma.
8. Call label scan routine.

This does not take into account, of course, the semantic aspects of the language (i.e. what to do with those constructs once they have been scanned), but that can be added without difficulty. Note the elegant simplicity of this kind of organization. It is not necessary for the IF statement to concern itself with what an expression or a label looks like. It does not have to look at the first character after the right parenthesis to see if it is a digit, and if so say, "Oh good, then this can be a label." It knows that label scan will take care of knowing what a label ought to look like, and give diagnostics if it does not find the right thing. Of course, this requires a careful definition of the various constructs, so that there is no ambiguity. For example, if labels were (somehow) defined such that they could contain commas, the above syntax would be ambiguous. Thus it is not completely true that the various levels can operate without knowing what the other levels do, but is nonetheless a guiding principle that this should be the case wherever possible.

Let's go further, and consider the syntactical breakdown of "expression".

expression = any number of terms, separated by "+" or "-".

term = any number of factors, separated by "*" or "/".

factor - any number of elements, separated by "**"/

element = variable or constant or function or "(" expression)".

variable = scalar or array element. scalar = identifier.

identifier = letter, followed optionally by letters and/or digits.

This hierarchal arrangement of the compiler automatically takes care of the hierarchy of arithmetic operators. Multiply will be done before add, because the term scan routine will process all the multiply signs it encounters before returning to an upper level to process plus or minus signs. (Term scan encounters the plus or minus sign, does not recognize it as something that it is responsible for, and so exits.)

Notice that expression scan goes down several levels before it even looks for at a character on the source line. The first three levels do not need to, because they each need one of the lower level constructs before they can proceed. (Actually, this is not completely true, because an expression may be preceded by a plus or minus sign, and that is not taken into account above, but idea is correct.) It is not until we get to the element level that we have to look at a character and decide what to do. If there is a left parenthesis, we simply come back in the top, by calling expression scan (recursively). This avoids having to keep stacks of parentheses and operators to keep track of what is going on. What we keep instead is a stack of return points (on the EXITLIST). If the first character is a digit, we can assume that the element is a constant. If the first character is a letter, then the element is a scalar or array variable or a function, and we can distinguish these by their previous declarations (e.g. DIMENSION, EXTERNAL) and by whether a left parenthesis follows the identifier.

Thus you see that we are introducing a bit of "bottom-up" procedure at this point. We look at a character (which is the absolute bottom), and then decide what to look for next. A truly bottom-up compiler would not have the hierarchy of levels (term, factor, etc.) that we have.

To scan an expression, it would start right off by looking at the first character, and that would tell it how to scan the first element. Then it would be prepared to see any of the operators (+, -, *, /, or **) and start stacking operands in the right places to handle the operator hierarchy.

On the other hand, a truly top-down approach would never look at a character to decide what to do, and that is where the TRY POP comes in. A completely top-down compiler, when it got to element scan, would TRY a scan for constant, and if that failed, it would TRY a scan for array or function, and so on.

Obviously this would be extremely roundabout and time-consuming. Whenever possible, our compiler will look at characters to decide what it ought to do. It does so, for example, more than the 900-series FORTRAN II. That compiler was so top-down in its overall statement scan that it would consecutively try (with QSF, "Quote Scan or Fail", a POP we do not have) all the various statement quotes, with a FEX and FAIL in between each. On ours, we look at the first letter and then try the various quotes beginning with that letter (using QSA, which does not have to go through FAIL).

The tricky part of a top-down FORTRAN are those places where you have to get quite far into something before you can find out what it is. At the statement level, it is so bad that the whole FEX and FAIL procedure described in the previous sections is necessary. But even when you know what statement you are scanning, parts of it still require multiple scans. In FORTRAN II, there were very few such places. The TRY POP was used rarely; and never on long, time-consuming operations. The introduction of new language features into FORTRAN IV, however, has greatly increased the number and severity of ambiguous constructs that require multiple scans in order to be handled in a top-down environment. Let's look at a few examples:

1. Complex constants. Previously, a left parenthesis within an expression meant a nested subexpression, but not that is not necessarily so. Consider the following two expressions:

   ```
   A * (3.2 + B)

   A * (3.2 , 4.3)
   ```

   The first contains a subexpression, the second a complex constant, and you cannot tell which until you get to the plus sign or comma. Suppose, after scanning the left parenthesis, that you see the next character is a digit, so you decide to scan for a complex constatnt. When you get to the plus sign, you realize you were wrong, but it is difficult to recover, because you are in the wrong place. You do have the first element of an expression, but you have not come down through expression scan, term scan, factor scan, and element scan to get it. You cannot just branch into the middle of factor scan, because the EXITLIST is not properly set up to take you to the right places to process this element. You either have to make major adjustments to yout top-down scheme, to make it more bottom-up, or you have to do a multiple scan. We do the latter. We TRY complex constant, and if that fails, then we assume it must be a nested subexpression.

2. Logical expressions. Consider the following lines, each of which might begin an IF statement:

   ```
   IF ( A*B )

   IF ( A*B C )
   ```

Not only are the two expressions of different mode, but until you knew what mode they are, you do not even know which kind of IF statement you are dealing with. And you do not know the mode until you see either the right parenthesis or the greater=than sign, and again it is too late. The logical expression scan has its own hierarchy (EQR; OR; AND; NOT; relational expression; arithmetic expression), and if you scan the A*B as an arithmetic expression, then when you find the greater-than, you ought to be in the middle of relational expression scan, buy you are not.

3. Logical and relational operators. The syntaz chosen for these operators makes them ambiguous when they follow integer constants. Consider the following examples:

```
IF ( 4.EQ.X )

IF ( 4.E9-X )
```

The first is logical, the second arithmetic. Constant scan certainly has no business scanning three characters into a relational operator; that is supposed to be handled several levels up in the hierarchy. But it has to, in order to find out whether ther is actually an exponent there, not an operator. And it has to use a TRY so that, if it is an operator, it can back up and leave it for the proper level to handle.

4. Expressions in output lists. Whereas previously a left parenthesis in an I/O list always indicated another nested I/O list, now it may indicate a nested expression, or worse yet, it may be only one element of an expression, as shown below:

```
( B(I) , I=1,10 )

( B(I) + C )

( B(I( + C ) * D
```

The first is a nested I/O list. The second can be treated as a nested I/O list with no ill effects (a nested I/O list does not have to have DO-control). But the third cannot. It comes back, after the second right parenthesis, with a perfectly good nested I/O list, but it is totally unprepared to cope with the asterisk that it finds next. We ought to be partway down the expression hierarchy, in term scan, to handle that asterisk. On the other hand, if we assume that the construct is an expression, it will handle the second and third cases above just fine, but will not be able to do anything with the comma following B(I) in the first case. Here again, this has to be done with a TRY.

There are other examples of this kind of thing, but the above are representative, and two of them, logical expressions and expressions in output lists, are the worst offenders (not to mention logical expressions in output lists!). What has happended is that the language has gotten to a point where a top-down scan is just not efficient for some constructs. The result is that

some statements require a good deal of multiple scanning, with corresponding loss of speed. What would be needed is to retrench slightly towards bottom-up, so that the above constructs could be dealt with as you go along. Almost none of them cuase serious syntactical problems such as the "DO3K=1" that was described under multiple scanning of statement classes. It is mainly a problem of being able to do the right thing with what you have, once you find out what it is. This change in technique could probably be added to the compiler without completely disrupting it, but it would require some study.

2.7.4 The TRY POP

TRY is sort of a tentative JRS. It is a JRS if it succeeds, or a NOP if it does not. One can TRY a certain scan, and if it does not fail, then it returns, just as if the TRY had been a JRS. However, if anything drastic happens (such as illegal syntax), and control reaches FAIL, then special action is taken. Instead of going to FEXFAIL (as described in previous sections) and scanning for a different kind of statement, control is returned to the TRY POP itself (via TRYFIAL), which restores everything (scan pointers, lists, etc.) to the way it was before the TRY was executed. TRY also always returns an answer. If it succeeded (i.e. acted like a JRS), it returns with answer TRUE. If it failed (i.e. acted as a NOP), it returns with answer FALSE.

It is the responsibility of any TRYed routine to save any lists that it may add to, except for the EXITLIST, WORKLIST, and ALPHALIST, which are automatically saved by the TRY POP. SYTCHLST type changes are automatically restored by TRY. Note that this necessity for saving lists means that it is not usually practical for the same routine to be reached both by JRS and TRY. If this is needed, the TRY will enter just above the standard entry point, save the lists and fall into the standard JRS code.

There is another feature of the TRY POP that does not exist under FEX. That is the Automatic Succeed Flag, mentioned previously as being the bottom word on the TRYLIST. TRY does not do character count comparisons (i.e. with OLDCHCT and NEWCHCT) to see which of various TRYs got the furthest. Either a TRY succeeds or it fails. Sometimes, however, it is useful to make it do a little of both Consider the statement:

```
CALL ALPHA ( A+B .GT. C .AND. .NOT. #S%/8)
```

We TRY logical exprssions before arithmetic expressions in a case like this (where iether type is permissible), because the former is a superset of the latter (a logical expression may contain arithmetic expressions as elements of a relational, as indeed the above case illustrates). The TRY fails because of the # character, but it would be misleading for the TRY to come back FALSE (i.e. saying it did not find a logical expression). It is clear that the user did intend a logical expression, and if we scan it now as an arithmetic expression, it will fail on the first period (or possibly the G), with a dianostic that is not the least bit meaningful to the user.

Therefore, whenever a TRYed routine gets far enough that it is convinced that what it is seeing really is what it is looking for (even though a later error might cause it to fail), it sets the Automatic Succeed Flag. Note that this is not very top-down. What it is really saying is that the construct looks more like this particular thing than like any other thing that might be TRYed, and it should not have to know about that, but it is an effective compromise to obtain better diagnostics.

When the Automatic Succeed Flag is set, TRY will never restore things and return FALSE. Instead, it will act just as a JRS would have acted. That is, if control got to FAIL (something drastic happened), the TRY will go right back to the FAIL at its level, i.e. a statement level fail (unless it is a TRY within a TRY, which is ghastly to consider but can happen). If there is a delayed fail error lurking on the TRYLIST, TRY just passes it up to the upper level without doing anything special about it.

Note that the TRY POP begins in PASS1ML, but immediately recurses into POP language and is continued in PASS1F. This is one of the few POPs that is itself mainly written in POPs.

The following paragraphs are a continuation of the paragraphs labeled A through J under Overall Statement Scan and Fail Structure, 2.7.2.

K. TRY

K.1 Save (FAILEXIT=TRYFAIL). (See steps A.1 and B.2). If a scan fails, this will cause FAIL to go to TRYFAIL (rather than FEXFAIL).

K.2 Call FEXTRYSV (paragraph C) to prepare for being able to fail.

K.3 Save the EXITLIST. This would be saved by FEXTRYSV, except it has to be saved at the same recursive level that you want it to return to. Note that the only three lists saved by TRY are the EXITLIST, WORKLIST, and ALPHALST.

K.4 JRS to the routine to be TRYed (see JRS, paragraph L). If it cannot get through its scan, it will go to FAIL (paragraph E), which will go to TRYFAIL (paragraph M because of step K.1.

K.5 If the called routine does get through its scan and returns from the JRS, test to see if the Delayed Fail Flag (on the TRYLIST) is set. If not, everything is fine, the TRY succeeded, so jump down to step K.8.

K.6 Test the Automatic Succeed Flag (bottom word of the TRYLIST). If it is not set, then the delayed fail that we found really means that the TRY did not succeed, so go to FAIL now, which will have the same effect as going from step K.4 (via paragraph L), except that we have proceeded farther through the scan. In general this is not a good thing for TRYed routines to do, because it does not result in any more diagnostics (they are all thrown away); it only takes more time.

K.7 Otherwise (i.e. the Automatic Succeed Flag is set), turn on the Delayed Fail Flag at the upper (calling) level. This makes the TRY do just what a JRS would have done. Note that since we want to set the upper Delayed Fail Flag before removing TRY's two words from the TRYLIST (only because it is convenient to common up the code for doing so), we have to get at it on the fourth word on the TRYLIST.

K.8 Remove the Automatic Succeed Flag and the Delayed Fail Flag from the TRYLIST. These were put on by FEXTRYSV (step C.3).

K.9 Unreserve the ERRORLST and the SYTCHLST. Since the TRY is going to succeed (i.e. act as a JRS), any errors it has collected, or changes made to lists, are valid -- at least as valid as if they had occurred under a JRS. We may still fail at an upper level, in which case the errors may be thrown out and the SYTCHLST changes restored.

K.10 Release the SAVELIST. As far as we are concerned, the TRY succeeded. We are not going to back up this scan and try anything else. The upper level might still fail, but it will have saved the lists so that it can recover. The SAVELIST was reserved in FEXTRYSV (step C.1).

K.11 Exit TRUE. (TRY succeeded.)

L. JRS Under TRY

Within the TRY POP is a JRS to the TRYed routine. This paragraph shows the things that may happen in this routine that are important in the TRY and fail process.

L.1 If, at any time during the scan, an error of severity 3 occurs, the Delayed Fail Flag (on the TRYLIST, see step C.3) will be set. This is done by the error POP (ERC, ERL, or ERW).

L.2 If, at any time during the scan, the TRYed routine decides that it has seen enough to convince it that this really is what it is looking for (even if a later fail error should occur), the Automatic Succeed Flag (also put on the TRYLIST by step C.3) will be set. This will be tested by step K.6 or step M.3.

L.3 If the scanning routine gets to a point where it is impossible (or undesirable -- this is often the case under TRYs; there is not much point in wasting time in a TRY if you know that it will not succeed) to continue the scan any farther, control will pass to FAIL (paragraph E). FAIL will go to TRYFAIL (paragraph M) because of step K.1.

L.4 Otherwise, the TRYed routine will exit back to the TRY POP. Either of the flags on the TRYLIST may be set.

M. TRYFAIL

M.1 Release the SAVELIST and the SYTCHLST. This information has been restored by FAIL, but FAIL cannot release the lists because, under FEX (rather than TRY), it may need them again.

M.2 Remove the Automatic Succeed Flag and the Delayed Fail Flag from the TRYLIST. Remember whether the Automatic Succeed Flag was set.

M.3 If the Automatic Succeed Flag was set (before being removed in step M.2), then we have to force the TRY to succeed, i.e., act like a JRS, so jump to step M.6.

M.4 Otherwise, the TRY has failed. FAIL has already restored all the lists and backed up the scan pointers. Now release the Error List since these error messages are not relevant.

M.5 Exit FALSE from the TRY POP.

M.6 The TRY has failed, but the Automatic Succeed Flag is set. This means an unrecoverable error has occurred, but we are not going to improve things by doing another scan; the diagnostics will just get worse. What would JRS do in this case? It would accept what it had done during the scan as being valid and then go to FAIL. It is too late for us to accept the list changes that may have been made; they have already been restored by FAIL. So we cannot proceed exactly like JRS, but that does not matter. All those list changes are going to be restored anyway when we now go to fail. What is important is that we retain the current error messages. So unreserve the Error List, thus combining the current error messages with those of the upper level.

M.7 Set INPCHCT = NEWCHCT. FAIL is probably going to go to FEXFAIL, which will want to know how far we got in this scan. However, the Input Character Count has been backed up (by the FAIL that just occurred) to be ready for another scan. We are not going to do another scan and we would like to get back the Input Character Count as it was just before the TRY failed. Fortunately, this has been saved for us in NEWCHCT (by FAIL, step E.2).

M.8 Go directly to FAIL.

2.8 Identifier Resolution

All identifiers, except for labeled COMMON block names, are kept in the Symbol Table. (Block names are kept in the BLNAMLST.) Each Symbol Table entry

contains six words. The first two contain the eight-character name, left-justified with trailing blanks. For the purposes of identifier resolution, the important words are the third and fifth. The third word contains the class (scalar, subprogram, etc.) in the first two hex digits, the type (integer, real, etc.) in the third hex digit, and traits or flags (such as dummy, intrinsic, etc.) in the last five hex digits. The fifth word is involved with "Permissibility" and is discussed in the next section.

The class for all identifiers starts out as UNUSED (X'E0'). Then when the correct class has been established, this is updated to the proper value. It is also possible for an identifier to remain UNUSED, e.g., by appearing only in a type statement. Identifier class can be set up in a variety of places. Arrays are set up in Allow Dimensions, part of the DIMENSION statement. Multiple Dummies are set up by a section under the FUNCTION statement (also used by SUBROUTINE and ENTRY). Subprograms can be set up by the FUNCTION, SUBROUTINE, ENTRY, CALL, CONNECT, and EXTERNAL statements. Most often, however, they are set up by appearing in an expression. This is also true of scalars, though a scalar can be set up by appearing in a DATA statement or a few other places (such as ASSIGN, DO). The class bits are set up by the REG POP, described below.

The type field is set up only by SETTYPE, which is part of Expression Scan. The various type statements (e.g. INTEGER, REAL) call SETTYPE to assign their particular type. Appearing in an expression can also determine the type of a symbol. Expression Scan calls REGTYPE, which calls SETTYPE. The type field is also set up with the REG POP, described below.

The traits (or flags) reveal various characteristics that are known about a symbol. Some of them are important only to Pass 2 (e.g. Addressable, Subscripted) and in fact, some of them apply more to other pointers than to symbols, but nonetheless their absence from the symbols may be significant. For example, no symbol will have the "Constant" trait, which is a useful distinction for Pass 2 to make. The traits are also set up by the REG POP, described below.

The traits are sometimes useful for helping resolve the class of an identifier, but the principal technique involved is the use of the Permissibility bits, which are contained in the fifth word of each Symbol Table entry.

2.8.1 Permissibility

Resolving the class of an identifier and (particularly) catching conflicts in declaration or use can be rather involved. There are many such uses and declarations that affect one another. To deal with this, one can take either a backward-looking approach or a forward-looking approach. Consider the statement

A = F(X)

What is the class of F? Well, if it has appeared in a DIMENSION statement, then clearly it is an array, and X is a subscript. Or if F had been previously determined to be a scalar, then this is an error, but can be treated as implied multiplication. If F has never appeared before, on the other hand, we will decide that it is a subprogram, and X is an argument. This is all based on looking backward (at previous usages) and presumably most compilers do this.

The trickier part comes with usages that affect, but do not uniquely determine, the class of an identifier. For example, suppose F had appeared in a COMMON statement. That does not determine whether it is a scalar or an array, but it is certainly not a subprogram. The same would hold true for F appearing in GOLBAL, EQUIVALENCE, or NAMELIST. Not only can usages affect the resolution of class, they can also affect other usages. For example, appearing as a dummy prohibits appearing in EQUIVALENCE, and vice versa. Appearing in a DATA statement prohibits appearing in blank COMMON, and vice versa.

How to keep track of all these interactions? The backward-looking approach is to look around and see if the identifier has been used in any way that would prohibit how it is now being used. For example, in the case of A=F(X), one can check to see if F has been placed in blank COMMON or EQUIVALENCE, etc. 900 Series FORTRAN II used this approach. The language was simpler and there were not too many things to check for. Still, it was a sequential check: Has it been used this way? Has it been used that way? And some of those checks were not easy. To determine blank COMMON, one had to search the blank COMMON list.

For these purposes, our compiler uses a forward-looking approach called permissibility. The fifth word of each Symbol Table entry is the permissibility word. It contains a set of bits, all of which are set to one when an identifier is first placed in the Symbol Table. Each bit corresponds to a permissible usage (everything is permissible initiallly). Whenever a name appears in a context that prohibits some other usage, the permissibility bits corresponding to the now-illegal usages are masked away. Thus, for example, appearance in EQUIVALENCE masks away the subprogram permissibility bit as well as others, such as Multiple Dummy, Intrinsic, etc.. This is all done in one quick operation, using the SUP POP (Set Up Permissibility), described below. Similarly, the testing is straightforward. The SOP POP (Set On Permissibility) tests for the presence of a particular bit in the permissibility word. Thus if we want to see whether a name can be a subprogram, we do a SOP SPROG, and that tells us whether the corresponding permissibility bit is still present. This is called forward-looking because instead of each usage having to know what <u>previous</u> appearances would make this usage illegal, each usage has to know what <u>subsequent</u> appearances are rendered illegal by this usage. This has the definite advantage of speed. It can have the disadvantage that, although you know that a current usage is illegal, because the permissibility bit is gone, you do not know what kind of appearance took the bit away. So you can give an IDENTIFIER CONFLICT error,

but you cannot explain to the user what pervious usage caused it. Actually, you could do this; we chose not to mainly for reasons of space in Pass 1. When an identifier conflict occurs, all the necessary information is present (in fairly convenient fashion) to do a backward-looking analysis of what caused the problem.

The following table shows the layout of the permissibility bits. The headings across the top show the positions of the SOP (testing) bits. For example, multiple dummy tests the X'80000000' bit; the DATA statement tests the X'00000001' bit. The headings down the left side are for the values of the SUP (extracting) words. For example, multiple dummy does not allow subprogram, array, or scalar, but it does allow all the types (logical, etc.). Most of the table is self-explanatory. A few additional notes will be useful.

Table 2.8.1 Identifier Permissibility

	MULTDUMY	SPROG	ARRAY	SCALER	LOCIGAL	KOMPLEX	COMPLEX	DOUBLE	REAL	INTEGER	SPROGDEF	INTRNSIC	EXTERNAL	DUMMY	GLOBAL	LABCOM	BLANKCOM	CALLED	BASICEXT	ASFDUMMY	NARGS	NAMEARG	NAMELIST	EQUIVAL	DATA
MULTDUMY	1				1	1	1	1	1	1								1		1					
SPROG		1			1	1	1	1	1	1	1	1	1					1	1	1			1			
ARRAY			1		1	1	1	1	1	1					1	1	1			1			1	1	1	1
SCALAR				1	1	1	1	1	1	1	1	1	1							1	1			1	1	1
LOGICAL	1	1	1	1	1						1	1	1	1	1	1	1	1		1			1	1	1	1
KOMPLEX	1	1	1	1		1					1	1	1	1	1	1	1	1		1			1	1	1	1
COMPLEX	1	1	1	1			1				1	1	1	1	1	1	1	1		1			1	1	1	1
DOUBLE	1	1	1	1				1			1	1	1	1	1	1	1	1		1	1		1	1	1	1
REAL	1	1	1	1					1		1	1	1	1	1	1	1	1		1	1		1	1	1	1
INTEGER	1	1	1	1						1	1	1	1	1	1	1	1	1		1	1		1	1	1	1
SPROGDEF		1			1	1	1	1	1	1			1					1		1			1			
INTRNSIC		1			1	1	1	1	1	1		1								1						
EXTERNAL		1			1	1	1	1	1	1	1		1					1	1	1			1			
DUMMY		1	1	1	1	1	1	1	1	1			1					1		1			1			
GLOBAL			1	1	1	1	1	1	1	1					1					1	1		1	1	1	1
LABCOM			1	1	1	1	1	1	1	1										1	1		1	1	1	1
BLANKCOM			1	1	1	1	1	1	1	1										1	1		1	1	1	
CALLED		1			1	1	1	1	1	1	1		1					1		1			1			
BASICEXT		1			1	1	1	1	1	1			1							1			1			
ASFDUMMY				1	1	1	1	1	1	1										1						
NARGS				1				1	1	1																
NAMEARG		1	1		1	1	1	1	1	1	1		1		1	1	1	1	1	1	1		1	1	1	1
NAMELIST			1	1	1	1	1	1	1	1					1	1	1			1	1		1	1	1	1
EQUIVAL			1	1	1	1	1	1	1	1					1	1	1			1	1		1	1	1	1
DATA			1	1	1	1	1	1	1	1					1	1				1	1		1	1	1	1
ASFUNC					1	1	1	1	1	1								1	1	1			1			

1. SPROG stands for subprogram. The SPROG SUP word is never used. The definition statements (FUNCTION, SUBROUTINE, ENTRY) SUP SPROGDEF instead (subprogram definition) and the usage (in expression scan) SUPs EXTERNAL so as not to allow intrinsic.

2. A lot of things disallow INTRNSIC, including CALLED. This means that currently we cannot have intrinsic SUBROUTINEs. This could be changed by changing the SUP word for CALLED.

3. Not many things allow DUMMY. This reflects the fact that dummies are not supposed to appear anywhere prior to their appearance in the dummy list. The exceptions are the type declarations.

4. LABCOM is labeled COMMON. It differs from blank COMMON only in allowing DATA.

5. BASICEXT stands for Basic External. Note that almost nothing allows it. A basic external function is generally recognized as such at its first usage. Even having a type declared prohibits basic external, because, after all, the only thing special about a basic external function is its type, and if that has already been declared, there is no reason to make it a basic external function.

6. Between the CALLED and BASICEXT SOP bits, there is an unlabeled bit that is never tested for. I think this was the ABNORMAL bit, which was never used.

7. NARGS stands for Number of Arguments and refers to the variable in parentheses after a multiple dummy. Note that its type can be declared later, but cannot be logical, double complex, or complex.

8. NAMEARG refers to appearing alone in an argument list. An array or subprogam can do this, whereas normally they could not stand alone without subscripts or arguments. The SUP word for NAMEARG is probably pretty superfluous. We mainly use the SOP bit in the routine that tries names alone as arguments.

9. Note that although DATA does not allow BLANKCOM and vice versa, it is still possible for a user to slip one past Pass 1 by putting data into a variable that has been EQUIVALENCEd to something in blank COMMON. This is caught later on in ALLOCATE.

2.8.2 Symbol Table POPs

There are six POPs that are used for setting up and testing identifiers. They are REG, SUP, SOP, SOT, SOM, and SOF. Each of them will address one of the entries in a table contained in the assembly LISTDEFS. This table is headed, "Class, mode, traits, and permissibility". Each entry in this table can contain three words. The first contains class, mode (type), and trait information and is used by REG, SOT, SOM, and SOF. The second contains only the one bit used by SOP for testing permissibility. The third contains the extract mask used by SUP to remove bits that are no longer permissible. Each of these three-word entries has a name attached to it (and externally DEFed). This name is used by all of the above POPs, even though some of them do not actually address the first word in the entry. For example, the instruction REG SCALAR does address the word that the label SCALAR is attached to. However, SOP SCALAR addresses the second word of that entry, i.e. it actually addresses SCALAR+1. This is all handled automatically by the POPs themselves, so the compiler code itself is quite simple and mnemonic. You do not have to remember to say "Set On Permissibility of SCALAR plus one".

Within the first word of each entry (sometimes called the REG word), there are three fields. The first two hex digits are for class and are used by REG and SOT. The next hex digit is mode (or type) and is used by REG and SOM. And the last five hex digits are traits, and are used by REG and SOF. Any one entry will be used by only one of SOT, SOM, and SOF. Some entries are not used by certain of the POPs. For example, there is no REG word for EQUIVAL, only permissibility words, whereas there are no permissibility words for LOCALVAR, only a REG/SOF word. This is allowable; the table is not dependent on there being three-word entries. Each of the entries is an independent entity, and each knows how many words it should have.

REG REGister

REG modifies the third word of the Symbol Table entry pointed to by WO. It uses the first word, i.e., the REG word of the 3-word entry addressed. The low-order six hex digits, i.e. the mode and trait fields, are simply merged into the Symbol Table word. The trait or mode may or may not have been there before; that does not matter. The first two hex digits (the class field), however, are treated somewhat differently. This is a direct replacement instead of a merge. If the addressed word has any bits set in the class field, the whole field becomes the new class field of the Symbol Table word. This was done because the class field is an entity unto itself, and also because the initial value of the class field (UNUSED) already has some bits in it. However, it seems as though a straight merge would actually work right, since all the other classes happen to contain the UNUSED bits as a subset.

There is one peculiarity with respect to REG (also SUP and SUL). It is used mainly during scanning of source statements and, as described under TRY-Fail, a scan may fail and be tried again. In this case,

any changes made during the scan have to be replaced. Therefore, if the new Symbol Table word created by one of these POPs differs from the old one (it may not; it may be registering a trait that was already there), then we have to save the old value and its location (relative to the Base of the Symbol Table) on the SYTCHLST (SYmbol Table CHange LiST). These can then be restored by the FAIL routine if necessary.

SUP Set Up Permissibility

SUP does an extract on the permissibility word of the Symbol Table entry pointed to in W0 (the permissibility word is the fifth word), using as an extract mask the third word in the three-word entry addressed by the POP. That is, the POP actually addresses the REG word, but adds 2 to the address to get the SUP word. This eliminates from the permissibility word those bits corresponding to usages that are no longer legal. Also, as with REG, the old value is saved on the SYTCHLST if it differs from the new one.

SOP Set on Permissibility

SOP performs a logical AND between the permissibility word of the Symbol Table entry pointed to by W0 and the second word (i.e. the SOP word) of the three-word entry addressed. It does not store this value back into the Symbol Table; it tests the result. If the result contains any 1's it returns with answer TRUE (i.e. this usage is permissible because the corresponding bit was there). Otherwise, it returns with answer FALSE.

SOT Set On Test (really means Set On Class)

Returns with answer TRUE if the ID field of the pointer in W0 is the same as the word addressed by the POP (i.e. the REG word). Otherwise, the answer is FALSE. For example,

```
SOT SCALAR
```

tests whether the symbol has previously been REGed as a scalar.

SOM Set On Mode (or Type)

Returns with answer TRUE if the pointer's type field is the same as that of the word addressed. For example,

```
SOM INTEGER
```

SOF Set On Flag (or Set On Trait)

Returns with answer TRUE if the bit (or bits) present in the word addressed (i.e. the REG word) are also present in the symbol's flag (or trait) field in its third word. Whereas SOT and SOM test for

complete equality of the fields, SOF tests whether any of the specified bits are present. Almost always SOF only tests for one bit, but it can test for more, such as in the case of

```
SOF  ALOCATED
```

which simultaneously tests for any of the traits corresponding to Blank COMMON, Global, equivalenced to Global Labeled COMMON, and Local. (Some of those traits are set up by Pass 1, others by ALLOCATE.) This POP also works on plex pointers.

2.8.3 Type or Mode Resolution

As explained above, the various types (integer, real, etc.) can be set up with REG, tested with SOM, and also have permissibilities that are set up by SUP and tested by SOP. When one type has been set up, for example, the way we recognize that we should not allow it to be declared some other type is not by doing a SOM to see if it already has some other type, but by doing a SOP on the new type. If it was already declared some other type, the permissibility of the new type will have been extracted out.

There are four main rotuines that handle the setting up and testing of type for all areas of Pass 1 (i.e. during scanning). After Pass 1, all the types have been establihsed. Note, however, that UNKNOWN (no type) is a legitimate type and can come out in the summary.

2.8.3.1 Register Type REGTYPE

This is used to give implicit type to a symbol when it appears for the first time and has not had a type declared In particular, this is called by Element Scan within Expression Scan when identifiers appear within expressions.

If the type is already known, we do nothing. This may be because the type was declared in a type statement or it may be because the type was already established by a previous call on REGTYPE. If the type is not know, we extract the first character of the identifier and get its entry in the Character Translate Table. This entry contains, in the mode field (third hex digit), the type that should be given to this identifier. This type was either set up by an IMPLICIT statement or was initialized originally according to the "IJKLMN" rule (see Character Translate Table, 2.4).

The new type is attached to the identifier by calling the routine SETTYPE, below. In addition, we have to update the Character Translate Table word by merging in the TYPEUSED flag bit. This reflects the fact that the implicit type has now been used and cannot be changed by an IMPLICIT statement (see IMPLICIT Statement, 3.56).

2.8.3.2 Set Type (SETTYPE)

This is used by REGTYPE and also by the type statements (e.g. INTEGER, REAL, ...). A type has been selected for the identifier pointed to by WO. This may or may not be a legal type for it to have. We call Impose Type If Possible (IMPTYPE) which will REG the type if it is permissible to do so; otherwise it will do nothing, but it returns with an answer whether it succeeded or not. (A conflict can occur only in type statements, not in expressions.) If it failed, we give a fatal type conflict error message, but continue processing anyway. This will be picked up as a delayed fail (see TRY-Fail, 2.7).

Note that all the work is actually done by IMPTYPE, but it is also used by places (e.g. Inherit Type, below) that do not absolutely require the type to be REGed; they just want it done if it is a good idea.

2.8.3.3 Impose Type If Possible (IMPTYPE)

This is used by Set Type and by Inherit Type. It is given a type in PO (what this means is a pointer to one of the three-word entries in the Class, Mode, Traits, and Permissibility table; it is going to use all three words). It is also given a Symbol Table pointer in WO. It checks for the permissibility of the given type and exits FALSE if not permissible. Otherwise, it sets up the permissibility of that type (with SUP) and REGs the type. Note that it does all this without knowing what type it is working with. Note also that it always addresses the same address (pointed to by PO), even though it is using the REG word, SOP word, and SUP word. This is all handled automatically by the POPs.

There is one other thing that needs to be done if this identifier is an array. To know the full size of an array, it is necessary to know not only its dimensions, but also how many words are allocated for each element of the array. This depends on the type of the array. An Integer array has an element size of one; a Double Complex array has an element size of four. The element size is set up by a routine called Set Element Size. If the type is known before the dimensions are declared, then Allow Dimensions calls Set Element Size. However, if the dimensions are declared first, then we have to wait until we know the type before we can set the element size (see Allow Dimensions 3.37). And now is the time that the type is known, so we do it.

IMPTYPE always returns with the answer whether it was able to impose the desired type.

2.8.3.4 Inherit Type If Possible (INHTYPE)

This is used in a few places that want to take the type from one pointer and inherit it to another. The new pointer is always a symbol pointer. The old one usually is, but in the case of intrinsic functions, the old pointer is an entry in the INTRLIST (see Process If Intrinsic, 3.10.1.3.4.3). Type is inherited from one identifier to another in the ENTRY statement and in statement function definitions.

The old pointer is assumed to be in the cell called TYPETEMP. If it does not have a type, we exit TRUE (we inherited the type that was there, namely none). Otherwise, we use that type to ADR an entry in the Class, Mode, Traits, and Permissibility Table. Then we call Impose Type If Possible, which will transfer the new type if it is permissable to do so, and will tell us whether it did or not. We return this answer to the calling routine.

2.8.3.5 Set Element Size (SETELSIZ)

Used by IMPTYPE and also by Allow Dimensions to set up the element size of an array as soon as both its dimensions and its type are known. The fourth word of an array's Symbol Table entry contains a pointer to the Array List. The third word of the entry on the Array List contains a pointer to the Group List. And the second word of that entry is going to contain a pointer to the Integer Constant List (the value 1, 2, 3, or 4) when we are through. Using the type of the symbol, we pick up the element size out of the Element Size Table and store it into that second word of the entry on the Group List. We also have to save both the old value (which is zero) and its location on the SYTCHLST, in case of scan fail. Note that this is one of the cases where what is being saved on the SYTCHLST is not a Symbol Table value; it is a Group List value. Therefore the "location" that is saved is actually a pointer to the Group List word that was changed.

2.9 Overall Statement Scan Organization

This section essentially describes the high level structure of Pass 1, including the scanning of labels, statements, compound statements, and logical IFs, as well as the clean-up operations performed at the end of Pass 1. It does not go into the details of failing and rescanning, as this is covered in detail in Section 2.7.2.

The top level is the overall line control, which directs lower level routines to perform: initialization, label field scan, statement scan, checking for terminating DO-loops, checking for compound statements, releasing cards from the Input List, and finishing up when the END card is encountered. The actual scanning of statements must be done by a lower level routine, because that routine is used also by the Logical IF statement, to scan the statement that it controls.

2.9.1 Overall Line Control

As described in Section 2.3, the term "line" here may refer to an initial card and any number of continuation cards. This was a poor choice of terminology; "line" and "card" ought to be synonymous, and another word should be used for what we here call a line. (Especially since elsewhere we do use line to be synonymous with card; for example "Line Numbers", Section 3.14.)

After taking care of compiler initialization, the overall line control loop (at the beginning of the assembly PASSIF) proceeds as follows:

1. (NEXTSTA). Call LINEINIT to do line initialization. This subroutine is only called once; it is a subroutine only to isolate the initialization code and avoid cluttering up the main loop. The line initialization is partly saving of information for the TRY-Fail process (see Section 2.7.2) and partly setting up triggers that need to be tested during the processing of statements and lines.

2. Call LBLFLDSC to scan the label field (see Section 2.9.5). If Label Field Scan fails, or if it processes an F in column 1, this will transfer control to LINEFIN, step 5 below.

3. Call STASCAN to scan a statement (see Section 2.9.2). This means a single statement, not a compound statement. However, if the statement is a Logical IF controlling a compound statement, that whole construct is considered a single statement. (You will note in the description of Logical IF in Section 2.9.2 that Logical IF calls Compound Statement Scan to get its controlled statement.)

   If Statement Scan is unable to find an acceptable statement, it will return with answer FALSE, in which case we will go to FAIL, which will eventually lead us to LINEFIN, step 5 below. The reason that STASCAN does not just fail itself is that, when scanning compound statements, we want to be able to go on and scan other statements even though some of them are illegal.

4. Having found an acceptable statement, call PROCLABL to process the label (if any) that was found by LBLFLDSC. This means checking for loops that may end on this statement. (See Section 2.9.6).

5. (LINEFIN). Control comes here regardless of whether the statement succeeded. First we release all the information that was saved by LINEINIT, step 1 above.

6. If there is a semi-colon next (not followed by end of line), call COMSTASC to scan for a compound statement, i.e. any number of statements separated by semi-colons (see Section 2.9.3). Note that we have already terminated any DO loops on the first statement of a compound statement. Compound Statement Scan returns with answer FALSE if any of the statements it finds fail, but everything is taken care of below; we can ignore the answer. Only Logical IF is interested in that answer.

7. Call RELLINE to release the current line from the Input List and, if appropriate, list it and output it for later use by Pass 3. (See Section 2.3.1).

8. Call FILECODE to output the Plex and Code Lists onto the Output List. (See Section 2.9.8.)

9. If the NOTENDTR (Not END Trigger) was set by the statement(s) just scanned, return to NEXTSTA (step 1 above) to process the next statement. Otherwise the last statement was an END statement; go to FINISHUP (Section 2.9.4), which is the end of Pass 1 and will return to the main control in FORTRAN4.

2.9.2 Statement Scan STASCAN

Statement Scan is a subroutine, and is recursive, because it needs to be called not only by the overall line control, but also by Logical IF, which is already a part of Statement Scan. The initialization and saving of information done at the beginning of STASCAN are described in Section 2.7.2.

If the DEBUG option has been requested (run-time debug), we first file a special indicator, called a NULLSTNO, on the CODELIST and remember where we put it. This is a null indicator added for run-time debug because Pass 2 needs to know which interface items it receives are the beginnings of executable statements -- "executable", in the sense that Pass 2 needs to generate check-in code ahead of them. Since we will not know whether the statement is executable until after we have scanned it, we put this null statement indicator out first; then, if the statement turns out to be executable, we go back and change that indicator to an EXECSTNO.

If Label Field Scan found an S in column 1 (i.e. the SCARDTRG is set), we go off to SINCOLM1 to process in-line assembly language (see Section 3.57.1). It will return to one of the statement exits described below, just like any other statement, unless it fails (which is rare), in which case we have set up the FEXEXIT to go directly to ILLEGSTA (see below). There is no other kind of statement to do a multiple scan for.

Assuming it was not an S in column 1 line, we can now go about the business of trying to scan the statement. As described in Section 2.7.2, there are three classes of statements -- quote, assignment, and statement function -- and we want to try scanning for them in that order. To begin with, we look at the first character of the statement. If it is not a letter, it cannot be a legal statement, so we go to ILLEGSYN (Illegal Syntax), unless the character is an end-of-line, in which case we go try for a Null Statement (Section 3.64). This is not the same kind of null statement as described above; this refers to a blank card in the source input.

If the first character is a letter, but not one of those that begins any of the quote-type statements (as indicated by SOCing the STALFLAG -- see Section 2.4), we go directly to Assignment Statement. For example, there are statements that begin with W, but none with X.

Otherwise, we do a bottom-up type sieve to locate quickly the kinds of statements that might be present, based on the first letter. For example, "G" can be GO TO or GLOBAL; "I" can be IF, INTEGER, IMPLICIT, or INPUT; etc. These are tried in an approximately decreasing order of likelihood, until we reach the letters that are sufficiently uncommon as to not be worth checking for specially, namely L, N, O, and T. The statements beginning with these letters are simply checked for by their quotes, one after another. In the same way, the various statements beginning with "G" or "I" or whatever are checked one after another; e.g., if it is not GO TO, try GLOBAL; if it is not GLOBAL, give up and go to assignment statement.

If we go off to scan for a quote type statement, we may or may not find an acceptable one. If not, we will back up and try assignment. If that fails, we back up and try statement function. In any case, we either find a legal statement or we do not. If we do, it will transfer control to one of the statement exits described below in Section 2.9.9. This will go through STATEXIT and STAFINAL and exit TRUE to whoever called STASCAN. If we did not find a legal statement, control will end up at ILLEGSTA, which will exit FALSE.

How the various statements are scanned is described in Chapter 3. The only important one here is Logical IF, which has to recursively call STASCAN. It saves some information, and changes some, so that STASCAN can operate in the limited environment of only being the controlled statement in an IF and not a whole statement standing by itself. This is described in Section 3.55.1.5. Then Logical IF calls Compound Statement Scan (Section 2.9.3), which calls Statement Scan again; this is now a recursive call. In this case, STASCAN goes through the same scanning procedures and exits to COMSTASC, which exits to Logical IF, which then goes to one of the statement exits, which will now function at this level.

2.9.3 Compound Statement Scan COMSTASC

This is called by the overall line control, when a semi-colon is encountered following the first statement on a line. (Actually, if the semi-colon is followed by an end-of-line, we recognize that this is a dangling semi-colon and does not really introduce a compound statement. The same is true on subsequent parts of a compound statement. This is handled by the subroutine CSASEMI, rather than just doing a CSA SEMCOLON.) COMSTASC is also called by Logical IF, to process the single or compound statement to be controlled. For this reason, it has to work recursively, which accounts for the added complication.

Basically, COMSTASC calls STASCAN over and over, as long as it keeps finding separative semi-colons (i.e. non-dangling). STASCAN returns an answer whether it found a legal statement or not, and COMSTASC keeps track of this cumulative answer on the WORKLIST. If any of the statements fail, COMSTASC returns the answer FALSE to indicate this. Only Logical IF is interested in this answer, not the overall line control.

Certain statements (such as FORMAT) must have labels, and there are other reasons that it is useful for statements to know whether or not they are labeled. The first statement on a compound line may have a label, but that label does not really apply to the subsequent statements. Therefore we have to turn off the HASLABEL trigger. When a FORMAT, for example, wants to complain that it does not have a label, it usually puts its error marker in column 5 (this is set up as the LBLCHCT by Label Field Scan -- see Section 2.9.5). This would not be appropriate for a FORMAT statement in the middle of a compound statement (especially if there actually was a label in column 5), so we prefer to put the marker at the beginning of the statement. For this

reason, before each statement in a compound statement, we update the LBLCHCT to be the current character count, so that if this error should occur, it will be marked at a sensible place.

However, if we are under a Logical IF, we cannot just go destroying these two values, HASLABEL and LBLCHCT. They do have to be changed for the time being, but they will have to be put back so the Logical IF can use them. So we save them on the WORKLIST too. This is one of the reasons that it is extremely important for all statements to leave the WORKLIST clean when they are done. Not only is it messy to leave things around on the WORKLIST, but in a compound statement, it would be disastrous.

The one other thing we do here is to set a trigger (CMPNDTRG) indicating that we are now in a compound statement. Because of implementation problems, we had to prohibit certain statements (notably DATA) from appearing in compound statements (the Plex List gets messed up). This trigger takes care of that. This is an unfortunate restriction, and one that would be nice to get rid of, if a cleaner implementation of the DATA statement could be devised.

2.9.4 Finish Up (FINISHUP)

Control comes here when the overall line control determines that an END statement has been processed. Control can also come here directly from Label Field Scan when a premature control card (exclamation point in column 1) is found, i.e., the END card is missing. There are seven things that need to be done at this point before returning to FORTRAN4.

1. Call DEKLUDGE (Section 3.76.1.2) to replace the FUNCTION name, if any. If we are in a FUNCTION subprogram, the name of the FUNCTION, registered as a SPROG, has temporarily been clobbered in favor of the name registered as a scalar. We now want to put the SPROG name back.

2. ZAP the TRYLIST to remove the two-word cushion put there by Compiler Initialize.

3. Check for unclosed DO or REPEAT loops. If there are any entries left on the DOLIST, we pull them off (from bottom to top), tell Pass 2 to terminate them, and print an error message for each. Since these are immediate error messages (as opposed to the delayed error messages used in PROCENDO, Section 2.9.7), we are able to include in the message the statement number that the DO was supposed to terminate on, as well as the line number on which the DO statement itself appeared.

4. Call FILRET1, which may have to create and file a RETURN or STOP. We do not want programs to run off the end because the user forgot to write a STOP or RETURN or transfer or something. On the other hand, we want to avoid putting out superfluous endings if they are not needed. It turns out that we have to file an extra RETURN (STOP is the same; Pass 2 knows which it ought to generate) unless all three of the following conditions are met:

A. The preceding statement is a transfer of some kind (including RETURN or STOP).

B. The END statement does not have a label -- which would allow the user to jump to a point off the end of the program.

C. The program is not being compiled in (run-time) DEBUG mode. In DEBUG mode, at run-time, the user can transfer to any line number he chooses, which would include the END statement even though it does not have a label.

In case either of the above steps produced code on the Code List, we call FILECODE to get it transferred to the Output List.

5. Check for undefined labels. We do this by stepping through the Local Label List and the Non-local Label List, SOLing each label for the DEFINED trait. If there are any undefined, we begin an error message, which can then include up to ten undefined labels. If there are more than ten, we start a new message.

6. Step 5 was the last one that might produce something to be output to Pass 2. (It might produce error messages which, when LO is requested, are sent along as part of the intermediate file.) So now we can file the ENDNO, which signals to Pass 2 that this is really the last thing coming. We then call FILECODE again to make sure that all of these final items ends up on the Output List.

7. Final pass through Symbol Table. This has two purposes. The first is fairly straightforward, and that is to get all appropriate names onto the Name List if we are NAMELISTing everything. This can come about either because of a NAME LIST statement with no identifiers or because the DEBUG option is in effect.

   The second purpose is to process identifiers that have never been classified, or never given a type, and need to have this done. It turns out that it is not at all obvious when this needs to be done. Some names are truly UNUSED, e.g., a name that has appeared only in a type statement or only as a dummy and was never used. However, a name that has appeared in COMMON, but was never classified as SCALAR or ARRAY, must be classified as SCALAR so that Allocation will work right on it. Similarly, some names can actually have no type, e.g., SUBROUTINE names or dummies. But others, such as those that have appeared in COMMON or EQUIVALENCE, must have a type.

   It happens that the single test "SOP SPROG" (is the name permissible to become a subprogram?) handles all of these cases. If it is permissible, it may already be one, or it may be a dummy, but it certainly will not have appeared in COMMON or EQUIVALENCE, and it will not have to be NAMELISTed. If the name is not permissible as a subprogram name, then we register it as a scalar if permissible, NAMELIST it if permissible, and give it the appropriate implicit type if it does not already have one.

Then, because we have been using the REG POP, and it was designed mainly for use within scans that might fail and therefore saves data on the SYTCHLST (see Section 2.7.1), we ZAP the SYTCHLST to throw away that data.

2.9.5 Label Field Scan (LBLFLDSC)

This is called at the beginning of the overall control for each line. It scans for a statement number in columns 1-5 and/or a special character in column 1, and it makes sure that column 6 does not contain a continuation indicator.

First we initialize the HASLABEL trigger to FALSE; this will be changed if we find a label. Then, if we are in run-time DEBUG mode, we file a LINENRNO and the current card count (LINECNT). Note that in update mode under XOS, this number will be different from the line numbers being used by the rest of the compiler (see Line Numbers, 3.14).

Next we want to look for a label. We used to do this by stepping through the label field one character at a time (with the Use Blanks Trigger set), but this was pretty time-consuming on ordinary cards with no labels, so we now use a more efficient, if slightly confusing, technique. We pick up the Input Character Count (INPCHCT), which is now pointing to the last column of the previous card, and add five so that it points to column five of this card. We store this into LBLCHCT (Label Character Count) as a default value. In case we do not find a label, this will point to column five where, for example, it was mandatory to have a label (e.g. on a FORMAT statement). The we add two more, so that it points to column seven, and we leave this around in WO for comparison purposes.

Now we SOC the Digit Flag. This automatically moves us all the way over to the first non-blank character on the card, wherever that may be (it could even be on a continuation card, if the first card was completely blank). If this first character is not before column 7, we exit immediately (there was no label or anything in column 1) with answer TRUE, indicating it is permissable to scan the rest of the line as a statement.

If we are in column 6, then the digit zero is valid and can be ignored. A letter might be the beginning of a statement that the user accidentally began in column six, so we give a dianostic and proceed. Anything else gets us to Illegal Syntax. Note that normally, we will not see anything in column 6 because the lower level scan routines (Next Active Character and Next Input Character) will have handled this during scanning of the previous line. The only cases in whihc this can occur are:

1. As the first card in the program.

2. After a comment card.

3. After an S in column 1 card (which does not allow continuation).

If the character we encountered is not a digit, then we go down and process it as a special character in column 1 (see below).

Otherwise, we step along, accumulating digits and packing them into an integer value, until we either hit a non-digit or we get to column 6. If we get to column 6, then we save the Local Label List (in case the statement fails later on) and register the accumulated value on the Local Label List. If we hit a non-digit before column 6, it can only be a dollar sign and there had better not be anything else before column six. If this is the case, we save the Non-Local Label List and register the value as that. In either case, we also put the Last Active Character Count (LACTCHCT) into Label Character Count (LBLCHCT), so that the latter will point to the last character of the label. This is used later on for putting error markers there (e.g., if a DO loop ends here and it is not supposed to.) We test whether this label has already been defined, and if so, print a warning; if not, set the HASLABEL trigger to TRUE, and set up the label (with SUL) as now having been defined. Save a pointer to the label in CURRLBL. File a LABELNO and a pointer to the label. This is where the label definition gets sent to Pass 2.

Now, if we are still looking at column 6, we do the same testing as above to see whether we can possibly proceed anyway.

If, when we looked at the very first character, it was <u>not</u> a digit, then we have a whole separate process to go through, namely, processing column 1 characters. If this first character is not in column 1, there is an immediate Illegal Syntax Fail. If it <u>is</u> in column 1, then there are a number of things we have to look for.

If it is a comment card (C, *, $, delta, /, or check mark), we call Release Card (RELCARD) to dispose of this one card. Then reset the CONTTRIG (Continuation Trigger, which may have been set by SIREAD) and the PRINTRIG (which would only be set if this were the first card in the program), and go right back to do a label field scan on the next card. This means that although the upper level called us expecting us to scan the label field of one card, we will skip through any number of comment cards until we get to a card that <u>has</u> a label field.

If there is an S in column 1, see whether the S option was specified. If not, fail with illegal syntax. The reason that we require the S option is not just a carry-over from the 9300 (where, to save memory, we did not bring the S-card processor into memory unless this option was specified -- this technique is not convenient on Sigma). In-line symbolic code is very un-FORTRAN and very incompatible with other systems, and we want to make sure that programs that use this feature are clearly distinguished by their control cards. If the S option was specified, then we set the SCARDTRG so Statement Scan will know what to do with this card and go back and scan for a label, just as on a normal card except beginning in column 2.

If there is an X in column 1, increment NRXCRDS (used by Summary in Pass 3). Then, if the X option was specified on the control card, go back and scan the label field beginning in column 2 (as we did with S in column 1). If the X option was not specified, proceed exactly as with C in column 1.

If an exclamation point (the character used to introduce control cards) appears in column 1, what we do depends on where in the program we are when this happesn. If we have already seen at least one source line, then we assume the END card is missing, print a diagnostic to that effect, and get out. Getting out is kind of tricky. We do not really want to go back to the upper level that we came from. So we knock the current exit off of the Exit List and branch directly to FINISHUP, which will then exit correctly from Pass 1 (rather than back to the place that called Label Field Scan). If this is the first card we have seen, then the situation is probably something like this: the user has his FORTRAN programs out on a tape and he is trying to compile them. But he forgot that they are on the tape along with some control cards in front of them. It would be a shame to assume missing END card, when we have not had any program yet. Instead, what we do is reduce the line count (LINECNT) to zero and treat the card as a comment. Thus, any number of preceding control cards will get printed as comments ahead of the program, each with a line number of zero. The program then, assuming we eventually find one, will begin with line number one, which is appropriate. Note that this situation will not occur when SI and C are the same, because the Monitor will then take ever reading the control card. (Actually, it might happen with identical ddevice assignments for SI and C if an EOD card appeared right after the FORTRAN control card. I'm not sure.) See SIREAD 4.2.1 for further information on how an exclamation point gets into column 1.

If the letters I, D, or B appear in column 1, we give a diagnostic and proceed to scan for a label as if the character had not been there.

If F appears in column 1, we perform some unattractive code which ought to be removed, because F cards are too much ancient history now to be worth being compatible with. We give a warning message, and then we call External Scan (EXTERNSC), which is the same subroutine used by the EXTERNAL statement to scan its identifiers. Then we exit FALSE. This is the only case in which we ever exit FALSE from Label Field Scan, and it means "Don't bother to scan for a statement; it's already been taken care of."

2.9.6 Process Label PROCLABL

This is called by the overall line control, after a statement has been successfully scanned. It was made a subroutine partly to keep the overall line control short and clean, and partially because it seemed like there might be several functions it would have to perform. As it is, there is only one function, checking for DO-loops terminating on this statement, and most of that is done by the subroutine PROCENDO (see below), which is also only called once. However, as a subroutine, PROCLABL is able to return answers from several places.

If there were any DO-loops (and that includes REPEAT loops) terminating on this statement, we check the DOERRTRG to see if that is permissable. If not, we put out a warning indicating that we have generated the termination code anyway. Then there is an interesting and subtle situation that has to be checked for. It came up as a bug after the compiler had been out for some time. Consider the following statements:

```
      DO 12, I = 1,10

12    IF (I  0)  IF  (I) 13,14,13

13    PRINT 5, I
```

The Arithmetic IF sets the AFTERIF trigger; the PRINT statement therefore puts its label on the IFLIST; and Pass 2 thinks it can generate code for the IF that falls into statement 13, instead of having to branch to it. Actually, of course, it falls into the DO-termination code. So we have to make "end-of-DO" act like an intervening statement with no label. This is done by calling No Label After IF (Section 2.9.9.7). The reason this situation was not anticipated (and does not exist in some other FORTRANs) is that everyone knows you cannot end a DO loop on an Arithmetic IF statement. But by ending the loop on the Logical IF that controls an Arithmetic IF, we have accomplished the same thing.

2.9.7 Process ends of DOs PROCENDO)

If this statement has no label, or there are no open DOs (and this includes REPEATs), then obviously no loops end on this statement. If the statement does have a label, and there are open DOs, then we initialize ANYDOS to indicate no DOs yet, and start looking for DOs that might terminate on this statement.

Theoretically, if a DO is going to end on this statement, it has to be the most recently opened DO, which is the bottom entry on the DOLIST. If this is the case, we file an ENDDONO on the Code List, and throw away the entire three-word DOLIST entry. The label was used to identify this as the termination statement; the control variable is used for subscript optimization purposes during the loop (e.g., are we in a loop with control variable K?); and the line number is used only in printing error messages, as described below. Finally we signal that there has been a loop termination and go back and start this test all over. In other words, there may be more than one nested DO ending on this statement, and we should find them in the proper order as we move up the DOLIST.

As soon as we have determined that there are no (or no more) loops ending properly on this statement (i.e., they are the bottom entries on the DOLIST), we then have to find out if there are any improperly nested loops terminating here (i.e., a DOLIST entry other than the bottom). We do this by searching the DOLIST for a match with the current label. If none is found, we exit, returning as the answer whether any DOs were found.

If there is a match (and we know it is not on the bottom entry), then what we want to do is kind of interesting. We want to print an error message, of course, and we mark it on the statement number in the label field. (This position was saved for us in LBLCHCT by Label Field Scan, Section 2.9.5.) It would be nice if we could identify the line number of the offending DO statement, since we have that information on the DOLIST, but unfortunately this is a delayed error message (with a marker), rather than an immediate error message, and delayed error messages cannot have extraneous information inserted into them. They are recorded on the Error List, with a pointer to the message, and there might be several improperly nested DOs ending here, each with a pointer to the same text string. It seems likely that this procedure ould also have been done with immediate error messages, since the scan is finished and there is no possibility that the errors do not apply and will have to be retracted.

The other thing we have to do is get Pass 2 to generate code to terminate this particular DO loop, which is down in the middle of the DOLIST somewhere. Normally, Pass 2 automatically uses the bottom (most recent) entry on the DOLIST to terminate a DO. So we want to send it a special indicator (called ENDONTNO) and a count telling how far back the DO is on the DOLIST. This count begins at zero for the first DOLIST entry above the bottom one, and goes up by one for each entry thereafter. We compute this count by reserving the WORKLIST, then piling DOLIST entries over there until we come to one that matches our current label. We pull that one out and then put all the others back and repeat the search to see if there are any more. Note that the search will find the topmost matching entry, whereas piling them onto the WORKLIST from the bottom will find the bottommost. This does not matter. If there is more than one improperly nested DO ending here, we would just as soon terminate them starting with the most recent (as we do with properly nested DOs). The search only serves to tell us that there is at least one improperly nested DO somewhere, and as long as it keeps telling us that there is at least one, we will keep going through the procedure of finding one and taking it out.

2.9.8 File Code FILECODE)

This is called by the overall line control, at the end of each line. Its main job is to transfer the information accumulated on the Code List onto the Output List (see Section 2.12). Prior to this, it has to put the Plex List information onto the Output List, so that Pass 2 will have loaded this information back onto the Plex List before it tries to process the intermediate output from Pass 1, cinse this intermediate output makes use of pointers to the Plex List. Originally, this process consisted merely of seeing if there was anything on the Plex List, and if so, copying it onto the Output List, preceded by the indicator PLEXNO and the count of the number of words. With the addition of some loop-optimization code, however, this process has become somewhat more complicated. This will be described in more detail in Section 3.15.

In brief, we no longer unconditionally output the Plex List after each line. Inside a DO (or REPEAT) loop, we accumulate information on it over a number of statements. In FILECODE, therefore, if there is anything on the Plex List, it is either brand new information or it is accumulated. If new, we send Pass 2 the PLEXNO indicator as before. This tells Pass 2 to empty its Plex List and start from scratch with the information we are now sending. We also save the count of words on the Plex List, to use next time in determining whether this is new or accumulated.

If Plex information is being accumulated (i.e. PREVPXCT is not zero), and there is some new information that has been added (i.e. current plex count is greater than PREVPXCT), then we send Pass 2 a different indicator, namely PLEXACNO, which tells Pass 2 to just add this information onto its existing Plex List. This is followed by the count of new words and then the words themselves. The words are output by a machine language subroutine, ONEWPXML. Whatever was odne about the Plex List above, we now decide whether to accumulate Plex information for next time. If we are not in any loops, or if the loop level has changed since last time (indicated by counting the DOLIST and comparing with PREVDOCT), then we wipe out the accumulated Plex List, zero the PREVPXCT, and start over with a fresh Plex List. Ohterwise, we leave the Plex List alone. See also Plex Passing; 3.15.

Finally, and regardless of what has been done above, we copy the Code List onto the Output List and call DUMOUTML to see if this new data on the Output List needs to be dumped onto a scratch file (if memory previously overflowed).

FILECODE is also called a couple of times in FINISHUP (Section 2.9.4), when special information has been generated on the Code List and needs to be moved to the Output List. In this case, no Plex or DO processing will be needed.

2.9.9 Statement Exits

2.9.9.1 Fail If BLOCK DATA FAILIFBD

Used by statements that must not appear in a BLOCK DATA subprogram. To double up tests, this routine also calls FAILFINE, to fail if not at end of statement or if there is a delayed fail. If this is a BLOCK DATA subprogram (indicated by BLDATRIG), we set up an error message and go to FAIL.

2.9.9.2 Fail If Logical IF FAILIFLI

Used by statements that must not be controlled by a Logical IF. Also checks for end of statement and delayed fail by calling FAILFINE. If the LOGIFTRG is set, give a diagnostic and fail.

2.9.9.3 Fail if Not End or if Delayed Fail FAILIFNE

All statements must go through this test before exiting. If we are not at an end-of-line or semi-colon, go to ILLEGSYN (Illegal Syntax) and fail. If the Delayed Fail Flag (next to bottom word on the TRYLIST) is set, just go to FAIL; whoever set up the Delayed Fail Flag has also set up an error message.

2.9.9.4 Fail if in DO loop FAILIFDO

Used by statements that must not appear within a DO or REPEAT loop, to give an error message and fail if the DOLIST has anything on it. For some statements (e.g. FUNCTION, ENTRY), this is a sensible check to make. The main reason that this subroutine was added, however, was due to some unfortunate restrictions resulting from certain improvements in DO-loop optimization. Inside loops, we need to keep the plex list around from statement to statement, and certain statements (e.g., DATA and adjustable dimensions) were just too strongly dependent on the assumption that they could destroy the Plex List during their processing. And it was much more complicated than just being able to save the Plex List somewhere and then put it back; it would have required some major revisions to the processing of the Plex List. It was more expedient to disallow these statements within loops, but it is an ugly and embarrassing thing to admit to. If these optimization features had been designed in originally, this would never have happened.

2.9.9.5 No Path Test NOPATH

If the preceding statement was a jump (as indicated by the JUMPTRIG), and this statement does not have a label (i.e., HASLABEL is FALSE), then this statement cannot be reached so we put out a minor warning message. Since Label Field Scan (Section 2.9.5) found no label, LBLCHCT points to column five, which is a good place to mark the error. Note that this subroutine is not used by transparent statements, such as declarations; the JUMPTRIG stays in effect until some active statement comes along.

The FORMAT statement branches into this routine at NOLABEL, which gives the same diagnostic if any FORMAT statement does not have a label.

2.9.9.6 Check Label After IF CHKLAIF

Used by non-transparent statements (usually non-declarative) to aid Pass 2 in the generation of good code for arithmetic IF statements. Following each such IF statement, the AFTERIF trigger is set. Then, if the suceeding statement has a label, it is put on the IFLIST so that Pass 2 can look there and see if that label is one of the branch labels on the IF statement; if it is, Pass 2 can generate one of the branches to fall into this point instead of branching to it. If there is no label on this statement, then it will not be possible to fall into this point, we put a zero on the IFLIST. In any case, we always turn off the AFTERIF trigger.

2.9.9.7 No Label After IF NOLBLAIF

This is a special entry point in the above routine that unconditionally places a zero on the IFLIST. This is used only by DO loops that terminate on arithmetic IFs controlled by logical IFs to prevent the code from thinking it can fall into the next statement, when actually a DO termination is in between. This is a highly subtle situation that does not occur in most FORTRANs because you are not allowed to terminate a DO on an arithmetic IF. But if you can terminate a DO on a logical IF controlling an arithmetic IF, the same situation obtains. See also Process Label, 2.9.6.

2.9.9.8 Non-FORMAT Label NONFORLB

Used by all but the FORMAT statement. If the statement has a label, it is set up as "defined on executable" (DEFEXEC, using the SUL POP), so that if it is later referenced as a FORMAT statement, a diagnostic can be given. If the label has already been referenced as a FORMAT the diagnostic is given now, using the label position saved in LBLCHCT by Label Field Scan.

2.10 Miscellaneous Scan Routines

2.10.1 Comma Tests COMMAT1,COMMAT2, COMMAT3, COMMAT4)

There are four comma test routines. Each essentially does a "CSA COMMA" and returns with answer true or false indicating whether there was a comma. Having the comma tests, however, allows us to do a number of nice things that CSA COMMA would not do:

A. Diagnose but allow extra commas between items.

B. Diagnose but allow dangling commas at the ends of lists. Note that even one dangling comma would cause a CSA COMMA to give the answer true, and the scanning routine would then get a syntax error because it could not find another list item following the comma. We know, however, that even if there is a comma, if it is followed by the end of the list, it is as if there were no comma, and we answer accordingly.

C. Item lists differ as to what constitutes the end of the list (used above). Some lists are enclosed within parentheses, so the end of the list is signalled by a right parenthesis. Others end on an end-of-line or semi-colon (compound statement seperator). In order to give the "dangling comma" diagnostic, we have to know which character is to be considered the end of the list. If that character is a right parenthesis, we also scan past it, so the calling routine will not have to.

D. Suppose there is no comma present. This may constitute end of list (CSA COMMA would say it was), but only if the next character is an end of list character (as described above). If the next character is not the end of list, but we also know there was not a comma, then we probably have two list items with no comma in between. In some cases this is legal (e.g. between FORMAT specifications), so we exit with answer TRUE, just as if there had been a comma. In most cases this is not legal, so we give a "missing comma" diagnostic and exit also with answer TRUE, as if there had been a comma.

The difference between the four comma tests, then, is what end of list character they look for and whether or not it is legal to not have a comma between items:

COMMAT1	EOL or semi-colon	comma optional
COMMAT2	EOL or semi-colon	comma required
COMMAT3	right parenthesis	comma optional
COMMAT4	EOL or semi-colon	comma required

2.10.2 Excess right parenthesis scan (EXCSRPSC)

This routine is called following the scanning of a right parenthesis, to diagnose but allow excess right parentheses that the user may accidentally have written. However, care must be taken to use this only when you know that it is not possible that some upper level routine is expecting a right parenthesis. For example, in the statement

```
A = 3 * (B + AMAX(X,Y,Z,W))
```

the section that scans the function arguments (Argument Sequence Scan) must not look for excess right parentheses, because any such parentheses belong to

the assignment statement and are not excess. However, the assignment statement can (and does) look for excess right parentheses.

2.11 Pass 1/Pass 2 Interface

Information gathered by Pass 1 is delivered to Pass 2 in several ways. Some of it is accumulated knowledge about the program, which is gathered on various lists and kept around for the rest of the compilation (or at least until it is not needed any more). For example, symbols are collected in the Symbol Table, along with their attributes. Labels are collected on the Local Label List and the Non-local Label List. Constants are accumulated on the various constant lists, and kept there until Pass 3, at which time the ones that are actually needed are moved to the Literal List. Other accumulated knowledge is kept in various triggers or counts, such as the Main Program Trigger or the Number of Error Messages.

The sequential file of information that is produced statement by statement, and over which the second pass of compilation is made, is placed on the Output List by Pass 1. It is produced in variable-sized groups, each group representing one "statement". Actually, this "statement" may not correspond exactly to a statement in the source language, for three reasons:

1. Some source language statements produce more than one "interface statement" to be sent to Pass 2. For example, a READ statement in the source language can create a BCD READ interface statement, any number of I/O Item statements, and an End of I/O List statement. Each of these is considered a separate statement by Pass 2. It does not know, nor does it need to know, that Pass 1 considered this all one statement. Unfortunately, in the historical development of this system (910 to 9300 to Sigma), there was never a clear resolution of the terminology involved, so the ambiguity between a source language statement and an interface statement is not made clear in the listings, and some awareness of context is necessary to keep these straight.

2. Some source language statements do not produce any interface data at all: for example, declaration statements, CONTINUE statements.

3. Interface statements are transmitted to Pass 2 for things that are not source language statements. This includes label definitions, ends of DO loops, line numbers, Plex List loads, etc. Pass 2 essentially regards these as statements, and in a sense they are; some languages have a specific statement for ending a loop, for example. FORTRAN happens to do it by giving you the terminal statement number at the beginning of the loop. Pass 2 could not care less.

Each interface statement begins with an indicator number telling which kind of statement it is. Here again, the terminology is not consistent with respect to these indicator numbers. Pass 2 calls them indicators; Pass 1 calls them numbers. Indicator is the better word; Pass 1 gets into the bind of calling

them "statement numbers" but they have nothing to do with the statement numbers that appear in columns 1-5 of the source lines. A table of these indicator numbers appears near the end of STASCAN. It corresponds to a branch table in Pass 2 that determines where to go to process each statement. The numeric values of these indicators are not documented in this manual, because they should not be. The compiler is parameterized in those two places to run correctly regardless of what those values are. They can be changed by reassembling STASCAN and PASS2. (Of course they should have been parameterized in one central place, but this gets rather tricky when you have to create a branch table with ORGs externally referencing EQUs in another assembly. Even if it can be done, the readability goes way down.) Because it requires these two reassemblies, this set of values has never been updated, and it includes some obsolete, unused entries, like Protected Function (from the 9300) and RELEASE Statement (a feature that was designed for the 9300 and never implemented). No matter; everything runs fine; there are just a few wasted branches to P2ABORT in Pass 2.

The indicator number determines how many other words of information are in the group. Some statements send only the indicator, like ordinary RETURN. Some send a fixed number of words, like REWIND, which sends the unit number, or ASSIGN, which sends a statement number (i.e. label) and a variable pointer. Some send a variable amount of information, with a count telling how much, like FORMAT, which sends a variable length string of alphanumeric characters. Some that you might think would send a variable amount do not, like CALL, which can have a variable number of arguments, but it bundles them up on the Plex List into a dummy group plex, ties that to the subprogram name in a Function Plex, and passed only the Function Plex pointer. Of course, the plex data has to be passes sometime too, and it is passed as a "statement" with a variable size, but this may not happen until later (see File Code, 2.9.8, and Plex Passing, 3.15).

Pass 1 information destined for the Output List is usually accumulated first on the Code List and then copied to the Output List at the end of the statement. The principal reason for this is that the data sent to the Output List is irretrievable; once you put it there, it may go out to a scratch tape or RAD file (see Scratch Files, 1.4.7), and on occasion, it is necessary to retrieve information on the Code List before it becomes final. For example:

1. A statement may turn out to be illegal and have to be deleted. Often, however, information has already been created for transmission to Pass 2 before the error has been detected (or even after, since we often go on processing and catch the error via a delayed fail). Since this information is still available on the Code List, it can be thrown away easily.

2. Sometimes the data for Pass 2 has to be created out of order. For example, the implied DO loops in an I/O statement are encountered only after some of the I/O items have already been processed and output. In this case, it is necessary to go back and insert information ahead of what has already been produced.

At the end of Pass 1, the Output List contains a large file of data for Pass 2 and the Input List is empty. Pass 2 now needs to take its input from the Output List and put its output somewhere (like the Input List). As you can see by rereading the last sentence, this would be a little confusing. In addition, the dynamic list arrangement and handling is specifically set up assuming that output is being tacked onto the bottom of the Output List, and input is being taken off the top of the Input List (see Reassign Memeory, 1.4.9). So instead of copying everything from the Output List to the Input List, we perform a slight trick between Pass 1 and Pass 2. We call Move Output to Input List, which alters the actual list parameters (base, start, top, bottom) of the Input List to be the same as the Output List, and then ZAPs the Output List. Thus, in effect, the Output List turns into the Input List and a new Output List is created from scratch. References in Pass 2 to the Input List result in accessing the information that was put onto the Output List by Pass 1. There is some question whether this is as effective as it was intended to be. Theoretically the new Input List moves away from the Output List fast enough to make room for Pass 2's output, so it does not have to be reassigned often. Actually, it appears that Pass 2 is likely to start putting things on the Output List before it even takes anything off the Input List, which means the whole huge Input List will have to be moved right away, possibly several times, thus defeating the whole purpose.

It was mentioned above that, in addition to the OUTLIST, a number of other lists are used to accumulate information during Pass 1. Those lists are:

SYMTABLE	CKCONLST	CRELBLST	ASTLIST		BLCOMLST
ARRAYLST	HCONSLST	LOCLBLST	DPRSZLST	BLNAMLST	
GROUPLST	ICONSLST	NLCLBLST	IFLIST		EQUIVLST
	RDCONLST	NAMELST			LBCOMLST
	SCONSLST	REFLIST			
	XCONSLST				

As previously mentioned, the interface items have various sizes. The first word of an item is always a code number (the "indicator") specifying which kind of item it is; following this, there are usually a number of additional words representing additional information needed to generate code for the statement. The number of words following the code word is usually fixed for a given interface item. For example, "Assignment Statement" is represented by a two-word item: the first word contains the Assignment Statement indicator (3), and the second a Replacement Plex Pointer. A few interface items (like the ones used to pass Plexes or source lines, for instance) can pass varying amounts of information. The table below lists the interface item types, showing their indicator code numbers, the labels used by Pass 1 to gen them (via FAD) and the corresponding item size (a "V" appears for variable-length

items). Refer to the Pass 2 listing for a detailed breakdown of the interface words; each statement generator is preceded by a comment which shows the use of each word in the corresponding interface item. For instance, the comment preceding ASSIGNG (Assignment Statement Gen) reads:

1. Indicator

2. Replacement Plex Pointer

Table 2.11.1 Structure of Pass 1/Pass 2 Interface Items

Hex Value	Name	Number Words	Statement Type
01	ASSIGNNO	3	ASSIGN
02	ARIFNO	5	Arithmetic IF
03	ASNMNTNO	2	Assignment
04	BKSPACNO	2	BACKSPACE
05	DECODENO	5	DECODE
06	ENCODENO	5	ENCODE
07	GOTONO	2	Ordinary GO TO
08	ASNGONO	3	Assigned GO TO
09	IFSNSWNO	4	IF SENSE SWITCH
0A	IFSLITNO	4	IF SENSE LIGHT
0B	IFOVFLNO	3	IF OVERFLOW
0C	INPUTNO	2	INPUT Without List
0D	PAUSENO	2	PAUSE
0E	RDBCDNO	3	READ BCD
0E	ACCTAPNO	3	ACCEPT TAPE
0E	ACCEPTNO	3	ACCEPT
0E	RDINTPNO	3	READ INPUT TAPE
0E	RDCRDSNO	3	READ Cards
0F	RDBINNO	2	READ Binary
0F	RDTAPNO	2	READ TAPE
10	RDDRMNO	3	READ DRUM
11	RDDSCNO	3	READ DISC
12	RETURNNO	1	RETURN

13	STOPNO	2	STOP
14	REWINDNO	2	REWIND
15	not used		
16	SNSLITNO	2	SENSE LIGHT
17	WRBCDNO	3	WRITE BCD
17	PRINTNO	3	PRINT
17	PCHTAPNO	3	PUNCH TAPE
17	PUNCHNO	3	PUNCH
17	TYPENO	3	TYPE
17	WROUTPNO	3	WRITE OUTPUT TAPE
18	WRBINNO	2	WRITE Binary
18	WRTAPNO	2	WRITE TAPE
19	WRDRMNO	3	WRITE DRUM
1A	WRDSCNO	3	WRITE DISC
1B	LABELNO	2	Label Definition
1C	ENDDONO	1	End of Loop
1D	not used		
1E	CALLNO	2	CALL
1F	DOSTANO	3	DO
20	ENDFILNO	2	END FILE
21	FORMATNO	V	FORMAT
22	FUNCTNO	4	FUNCTION
23	COMPGONO	3	Computed GO TO
24	LOGIFNO	V	Logical IF
25	not used		

26	not used		
27	ENDIOLNO	1	End of I/O List
28	IOITEMNO	2	I/O Item
29	IODONO	3	I/O DO-Control
2A	SUBRNO	4	SUBROUTINE
2B	ASFDNO	4	ASFD (Statement Function)
2C	ENDNO	1	END
2D	REPFORNO	3	REPEAT "FOR"
2E	PLEXNO	V	Load Plex List
2F	ENDLIFNO	1	End of Logical IF
30	ENDONTNO	2	End of Illegally Nested Loop
31	SOURCENO	V	Source Line
32	CONECTNO	3	CONNECT
33	not used		
34	not used		
35	not used		
36	not used		
37	CODENO	4	In-Line Machine Code
38	not used		
39	REPWHINO	2	REPEAT "WHILE"
3A	BEGINDNO	1	Begin DATA
3B	DPAIRNO	2	DATA Pair
3C	ENDATANO	1	End of DATA
3D	DITEMNO	V	DATA Item
3E	not used		
3F	not used		

40	not used		
41	not used		
42	not used		
43	LINENRNO	2	Line Number
44	ENTRYNO	3	ENTRY
45	ASRETNO	3	Asterisk RETURN
46	BCDEENO	5	READ BCD With END=, ERR=
47	BINEENO	4	READ Binary With END=, ERR=
48	INPUTLNO	2	INPUT With List
49	PLEXACNO	V	Accumulate Plex List
4A	EXECSTNO	1	Executable Statement (Debug)
4B	NULLSTNO	1	Null Statement (Debug)

2.12 CODELIST Manipulation in Pass 1

Most of the time, the CODELIST is used in a very straightforward way in Pass 1. During the scanning of a statement, various pieces of information are placed on the list, preceded by an indicator number telling what this group of information represents. Putting something on the CODELIST we usually call filing. Most things are filed with the FIL POP, which simply puts the addressed quantity on the CODELIST. Other items are moved on with the MON POP. Others are copied on from another list. The preceding indicator number is usually filed with the FAD (File Address) POP, which files its immediate address.

At the end of each statement (or compound statement), the routine FILECODE is called, and it contains a part called CODEOUT. This part copies the CODELIST to the Output List, which is the actual interface list for Pass 2. The reason why the CODELIST exists, why items are not sent directly to the Output List, is that there are times when the information that has been filed on the CODELIST turns out later to be invalid or incomplete. There are only a few circumstances in which this occurs, and these are enumerated below.

1. The most obvious is that when a statement fails, we want to throw away all vestiges of it, including interface items that may have been filed (especially since these items may have been only partially completed when the statement failed). The CODELIST is one of the lists that is saved before each statement (see TRY-Fail), so anything that has been filed there will automatically get lopped off when the statement fails.

   This would not work with the Output List, because it is not guaranteed to stay in memory. When memory is about to overflow, the contents of the Output List will be dumped to a scratch file. There is the possibility that this will happen at any time, so once you put a word on the Output List, you cannot try to bring it back or change it, because it may already be out on the scratch file. This is essentially the reason why the following cases could not use the Output List either.

2. For the purposes of run-time debug, a special word, called a NULLSTNO, is filed ahead of every statement. If the the statement then turns out to be an executable one, then just before Statement Exit, we go back and change that word to an EXECSTNO. This tells Pass 2 to generate a statement check-in.

3. The REPEAT statement, when it scans for a "triple", uses the same subroutine that the DO statement uses. That subroutine scans for the control expressions, builds a triple plex from them, and files it. REPEAT does not want it filed. It wants to pile up the triples (and the non-triples) and build a REPEAT List Plex out of them. So it pulls the just-filed Triple Plex off the CODELIST and saves it elsewhere.

4. When an implied DO-control is used in an I/O list, it appears at the end of the list, following the list items that it is supposed to control. This means that the interface information for those items has already been

filed on the CODELIST before the scan comes to the DO-control part. However, the DO-control interface data has to go to Pass 2 ahead of the items it controls. So the routine IODOCTRL (I/O DO-Control) temporarily copies the data on the CODELIST (which was previously reserved for this reason) onto another list. Then it files the DO-control interface data. Then it copies the other data back onto the CODELIST.

There are also a couple of places that put information directly onto the Output List. One is Output Print Buffer (PBOUT) in PASS1F, which files the source lines that are going to be interspersed in the object listing. There is no chance that this is going to have to be recalled or changed, so it would be wasteful to put it on the CODELIST and then copy it immediately over to the OUTLIST.

The other place is the DATA statement, which is quite peculiar. Because it has the potential for generating enormous amounts of interface data (e.g., initializing a 5000 word array), it puts its DATA pairs directly onto the OUTLIST. This is done by Expand Data Pair Sequence, which is not called unless the statement succeeds syntactically. However, conflicts in type between the constants and the variables can still cause a whole DATA group to be deleted. This is managed even though the interface information is already on the OUTLIST and thus potentially out on a scratch file. The DATA statement keeps a separate list (the DATA Pair Size List), on which a series of indicators is placed. These indicators tell Pass 2 whether it ought to ignore a certain amount of interface data that it would otherwise process. This is a kind of delayed (to Pass 2) deletion of part of a statement.

2.13 CODELIST Manipulation in Pass 2

All of the code-generation POPs put their code on the CODELIST.

The most usual sequence of events for the CODELIST in Pass 2 is for code to be generated onto it, then the code is moved one word at a time from the CODELIST to the OUTLIST by Assemble Code (ASMCODE).

A common alternative to this procedure, used especially with the bottoms of loops, is to generate a file of code onto the CODELIST, then move it as a file to the Delayed Codelist (DCODLIST), and then at some later time copy the file back to the CODELIST and output it one word at a time through ASMCODE.

A thing that is done that is rather analogous to the loop-bottom manipulation is what is done with respect to subscripts. The code for the main body of the statement is generated; then, if there are any retroactive subscripts to be generated, the code for the statement is copied over to the DCODLIST, then the code for the subscripts is generated at the top of the CODELIST, then the statement code is copied back onto the CODELIST so that the subscript code precedes it. Then finally, the code is output through ASMCODE.

2.14 Code Generation

2.14.1 Statement Gen Loop PROCSTAT

Pass 2's statement generation loop starts at Process Statement (PROCSTAT), and is a simple three POP loop that calls Next Statement (NEXTSTAT) repeatedly until the END Trigger (ENDTRIG) is set, at which point End of Program Gen (ENDPROGG) is entered. ENDTRIG is reset at the beginning of Pass 2 and set by End Statement Gen (ENDSTATG), which does nothing else.

Next Statement Gen (NEXTSTAT) gets the next word of input (by calling NXINWORD; see below), which is the statement indicator number of the new statement, and falls into Statement Gen from WO (STGNWO). The latter entry point is made available for the purposes of Logical If Statement Gen (LOGIFSTG), which has to be able to drive individual statement gen routines, passing the statement number in WO (c.f., 3.55.2.5).

It is STGNWO that actually drives the individual statement gen routines. To prepare for the generation of statement code, it does the following:

- Initializes data for the temp allocation routines: Temp Count (TEMPCNT) is set to Initial Temp Count (ITEMPCNT), and Spare Temp (SPARTEMP) is set to zero; in this way, ordinary (i.e., non-Private) temps are re-used in successive statements;

- If the Clobber Index Trigger (CLOBXTRG) is set, we call Clobber Dependent Indexes (CLOBDEPX) to forget the contents of index registers containing loop independent subscripts (c.f., 3.3);

- We call Remember X-reg/DD-reg status (REMXDML) in order to preserve a copy of the beginning-of-statement register context for Script Gen's purpose (c.f., 2.14.4);

- And we test the statement number to make sure it is in the proper range.

Thus prepared, STGNWO now enters the appropriate statement code gen routine via the Statement Gen BRU Table (STGBRUTB).

Fundamentally, a statement gen routine works as follows:

(1) It takes input (the Pass 1/Pass 2 interface code, as described in 2.11) from the Input List; it does this by calling upon Next Input Word (NXINWORD, in PASSOF); this subroutine takes the next (topmost) word from the Input List and returns it in WO; if the Input List is empty, it is filled with a record from X1, the Pass1/Pass 2 scratch file;

(2) Via code generation POPs (see 2.14.2), the statement gen routine builds code on the Code List; initially, the Code List is empty, having been left empty by the previous statement's call on ASMCODE;

(3) Via the BIF POP (see ADRCALC, 2.14.3.1), "retroactive subscript" information is placed on the Script List for later handling by Script Gen; initially, Script List is empty.

Typically, the statement gen routine returns to STGNWO by branching to Statement Exit (STATEXIT), having built up two files of information: statement code on the Code List, and "retroactive subscripts" on the Script List. STGNWO calls Script Gen to dispatch the Script List contents, emptying that list and adding to the Code List contents; finally, it calls Assemble Code (ASMCODE; see 2.15) to move the Code List contents to the Output List, performing simple operations on constants and labels as it goes, and leaving the Code List empty for the next statement. Two alternate statement exit points are made available for simpler gen routines:

- Statement Exit/ No Script Gen (STEXNS) for gen routines that cannot produce retroactive scripts;

- Statement Exit/ No Script Gen or Assembly (STEXNSA) for routines that produce neither retroactive scripts nor code.

2.14.2 Expression Generation

There are basically two expression generators in Pass 2 -- one for arithmetic expressions and one for logical expressions. The logical generator is a two-pass operation (Simplify and Generate) and has very few entry points. The arithmetic generator is a one-pass operation and has many entry points (see 2.16).

These generators are described in considerable detail in Chapter 3 -- the arithmetic generator in 3.10.2 and the logical in 3.11.2.

2.14.3 Address Generation

In generating code to evaluate expressions, Pass 2 has called upon the various expression generators to gen code that reduces all values to "Addressable" quantities, such as constants, scalars, and subscripted arrays. As far as the statement gen routines of Pass 2 are concerned, Addressable pointers (those which have the ADRSBLFG trait) are the addresses of the quantities they represent, and are passed (to the BIF POP) as addresses of BIFed instructions. But, to the object code, addressability is not quite so simple: some index register setup code may be required prior to the BIFed instruction, or that instruction may have to have its indirect bit set. In general, the execution of a BIF POP may result in the generation of many words of code at the current point, as well as "retroactive subscript" evaluation code to be placed ahead of the code now on the Code List. The mechanism to do all this is contained in ADRCALC (Address Calculation Gen), called by the BIF POP. ADRCALC is also invoked by GTEAX, a subroutine called when it is desired to compute the actual address of a quantity and leave it in a given index register.

2.14.3.1 Address Calculation Gen ADRCALC)

ADRCALC is given two words: op in W0 (containing the instruction's op code, R-register field, X-register field, indirect address bit, and information telling whether this instruction has word or doubleword-addressing), and adr in W1 (a pointer to an addressable quantity). The routine's function is to gen any necessary index register setup code and make appropriate modifications to the indirect and index fields of op. ADRCALC does not put op on the code list; that is up to the calling routine. ADRCALC returns the modified op, modified adr, and an integer constant, addend, to be used as an address addend. Some of the things ADRCALC concerns itself with are the following:

1. It must remember index register contents; it does so in terms of an expression pointer (kept in the YINDWHAT rung of an XREGLIST item) and an integer constant (in YINDADND) representing the difference between the expression's value and the current index register contents. (See also X-reg Context Storage, 2.17.1).

2. Where subscript expressions differ by constants, a common index register can be used; thus A(J)+A(J-2)*B(J+5) requires only one index register, containing J.

3. The instruction's index-alignment type (word or doubleword) must be taken into account in relation to the type of data being addressed. Ordinarily, op's indexing alignment corresponds to the mode of the subscripted quantity; e.g., LW is used to load Integer and Real (1-word) values, and LD for double precision (2-word) values. But, occassionally, this is not the case, as with the use of LM (a word-addressing instruction) to access double complex (4-word) data, or LW (Word) to load the imaginary part of complex data (2-words). In cases like these, an appropriate multiple of the subscript expression is needed in the index register, and this is taken account of in the common subscript recognition scheme. Thus,

   A(4*J+2)+AIMAG(C(2*J))+CD(J-3)

   (where C is complex and CD is double complex) requires only one index register containing 4*J.

4. When subscript nesting gets too deep, requiring too many quantities to be held simultaneously in index registers, or when a particular subscript expression is too complicated to be evaluated directly in one index register, ADRCALC must provide for the calculation of such values ahead of the current statement code at a time when all the index registers and the accumulators are available. It does this by assigning a Private Temp to such an expression and placing the temp and expression pointers on the Script List (SCRPTLST); later on (at

end-of-statement time), SCRPTGEN (Retroactive Script Gen) will be called to gen the script evaluation code and place it ahead of the main statement code. Meanwhile, ADRCALC simply uses the Private Temp as though it contained the subscript expression, already evaluated.

The kind of setup code produced by ADRCALC, and the modifications it makes to op and adr depend upon (1) what the given address pointer is (constant, scalar, etc.), (2) whether it is nondummy, dummy (ordinary), or multiple dummy, and (3) when an index register is required, whether there is one which already contains the desired value (possibly off by a constant difference). adr may be any of the following:

3 - pointer to one of the constant lists

- pointer to a scalar

- pointer to an unsubscripted array name (this can only happen for a receiving sequence dummy, and this is the only case provided for by ADRCALC)

- Constant Subscripted Array Plex pointer (CONSUBPX) having two terms to represent A(n): array name pointer (A), and ICONSLST pointer (n).

- Fully Subscripted Array Plex pointer (FULSUBPX) having three terms to represent A(e+n): array name pointer (A), integer expression pointer (e), and ICONSLST pointer (n).

The following table shows the setup code generated in the various cases; it displays the modified instruction as BIF would generate it, just after the setup code ADRCALC gens. The modified instruction, as returned by ADRCALC, is also shown explicitly (as op, adr, and addend) in the box following the code. Note that the indirect bit and index field are in the op word, and are represented by * and X respectively.

Table 2.14.1 Address Calculation Generation

adr	nondummy	dummy	multiple dummy
constant or scalar, A	op A op = op adr = A addend = 0	op *A op = op* adr = A addend = 0	(not applicable)
unsubscripted array name, A	(not applicable)	op A op = op adr = A addend = 0	(not applicable)
CONSUBPX, A(n)	op A+ns op = op adr = A addend = ns	[eval,X ns/k] op *A,X op = op*X addr = A addend = 0	[eval,X n] LW,X *A,X BGEZ $+2 LW,X 0,X op 0,X or *X op = opX or op* adr = abs 0 or abs X addend = 0
FULSUBPX, A(e+n)	[eval,X es/k+d] op A+ns-kd,X op = opX adr = A addend = ns-kd	[eval,X es/k+d] [AI,X ns/k-d] op *A,X op = op*X adr = A addend = 0	[eval,X e+n] [AI,X n-d] LW,X *A,X BGEZ $+2 LW,X 0,X op 0,X or *X op = opX or op* adr = abs 0 or abs X addend = 0

Non-Indexing

Indexing

where

n is the constant part of a subscript expression

e is the variable part of a subscript expression

s is the data element size in words

k is the instruction's index alignment type: k=1 for word, k=2 for doubleword

d is index contents' addend (=YINDADND rung of XREGLIST item)

X is an index register number

eval,X exp+d denotes the code to evaluate the expression exp, offset by a constant d, in index register X

As noted in the above table, k usually equals s; in any case, k divides s exactly, so s/k is an integer. The eval code is shown as optional, since it is not generated if the desired expression (possibly plus a constant d) is already in some index register. The optional AI is genned only if its address is nonzero. ADRCALC distinguishes the cases by SOFing DUMMY and SOTing MULTDUMY, ARRAY (only for the receiving sequence dummy case), CONSUBPX, and FULSUBPX. The cases not requiring an index (those above the dashed line in the table) are singled out first. In these cases, no code is generated and the modifications to the instruction are trivial: '*' is tacked onto op only in the scalar dummy case, adr is not changed, and addend is nonzero only in the A(n) case for A nondummy. In the indexing cases, processing is done in three phases: (1) determine, according to the case breakdown, just what value is wanted in the index register; (2) do whatever is necessary to obtain an index containing the desired value; and (3) gen additional setup code for the dummy cases and make appropriate modifications to op and adr, and establish addend value. These processes are described in detail below.

1. The quantity we want in the index register is as follows:

adr	nondummy	dummy	multiple dummy
A(n)	(not applicable)	ns/k	n
A(e+n)	es/k+d	(e+n)s/k	e+n

The appropriate quantity is established in terms of an expression part, exp, and a constant, addend, as follows: (a) for a constant subscript n, we set exp = the pointer to n and addend = 0; for a variable subscript e+n, we set exp = the pointer to e and addend = n (its value); (b) if adr is not a multiple dummy, we multiply addend by s/k (an integer) and replace exp by a pointer to the expression e*(s/k). The latter is done in MUL2SIMP by creating a Product Plex of two terms: e, and (s/k) (as a

pointer to ICONSLST); MUL2SIMP takes into acccount special values of constant factors like 0 and 1.

2. The desired value is obtained in an index register in one of the following ways:

(a) We first make a search of XREGLIST and DREGLIST to see if exp, as determined above, is already available in a register; the rung tested YINDWHAT (YDDWHAT on DREGLIST), a pointer to the variable part of the register's contents. If exp is found to be in an index register, or in a usable (see below) DD-register, we use that register for the current application, subtract its constant offset d from addend, and generate no eval code. Note: a DD-register is judged (by the subroutine TSTDDIND) usable if it is to be used in a way which will not change its contents; i.e., only in cases where the AI is not to be genned.

(b) If exp is not already available in a register, we call GETINDEX to allocate a free index register and proceed as follows:

(i) We search for exp on the Retroactive Script List (SCRPTLST); if found, we know that code will be produced (physically ahead of the current statement) to evaluate exp and leave it in the Private Temp associated with exp on the SCRPTLST; in this case, we simply gen LW,X PTMP and place exp in YINDWHAT, 0 in YINDADND.

(ii) If exp is not on the Script List, but is evaluatable directly in an index register (i.e., if NEVALXFG is off and there is at least one more free index register available), then we set the register busy (via BUSYX), call GTACUINT to gen exp evaluation code, free the register (via FREEX), and place exp in the YINDWHAT rung and 0 in YINDADND.

(iii) If exp is not on the SCRPTLST and is not evaluatable in the index register, then we put it on the SCRPTLST, along with a Private Temp pointer which we allocate, and proceed as in (i).

3. Having obtained the desired value (possibly off by a constant) in an index register, we gen final setup code and modify op, adr, and addend as follows:

(a) Nondummy: we multiply the current addend value (ns/k-d) by k to obtain the final addend value, ns-kd; we apply the index specification X to op and leave adr unchanged.

(b) Ordinary dummy; we must have exact agreement between the index register contents and exp, so if addend is nonzero, we gen AI,X

addend and add addend to the YINDAOND rung for this index register; we specify indirect address and indexing in op, set addend=0, and leave adr unchanged.

(c) Multiple dummy: we do as in (b) to insure exact index register contents, then generate

```
LW,X *A,X

BGEX S+2

LW,X 0,X
```

We now know that the correct address is in X, but we not know its value formally, so we set YINDWHAT to zero to "forget" its contents. If op is a word instruction (k=1), we specify indexing by X in op and set adr to a pointer to absolute 0. If op is doubleword (k=2), we set the indirect address bit in op and set adr to a pointer to absolute X (the index register number). In either case, addend is set to 0; thus the modified instruction is either "op 0,X" or op *X".

2.14.3.2 Guarantee Effective Address in Index GTEAX

GTEAX5 and GTEAX7 are subroutines used by dirivers like Function Call Gen (FCALLGEN) and Format Reference Gen (FORMGEN) who need the effective address of a quantity computed and left in a specific index register. These subroutines are called with an addressable expression pointer in W0; each one FETs and XREGLIST pointer to the corresponding index register and enters GTEAX, the general address evaluation subroutine.

GTEAZ first stores the X-reg pointer in NEXTINDX to force the selection of that particular register next time GETINDEX is called to allocate one. Now it FETs a dummy instruction that will appear to ADRCALC as a word-addressing instruction, and then calls ADRCALC to generate the index register setup code for the given quantity, as described in the previous section. The dummy instruction is not genned by ADRCALC, merely modified; by sensing those modifications, GTEAX determines what must be done in order to obtain the effective address. First, if the index field is nonzero (=Y, say) and different from the given register number X (this could happen, though not typically: the value might have been already present in register Y), then we gen LW,X Y. Now we gen the addend value returned by ADRCALC; it will be applied to the address of the next instruction. Finally, we generate one of the following instructions (with the address adr), depending upon the indirect and index specifications placed on op:

not indirect	indirect

not indexed	LI,X	adr	LW,X	adr
indexed	AI,X	adr	AW,X	adr

(AI is generated only if adr is nonzero).

2.14.3.3 Free/Busy an Index Register FREEX, BUSYX

Each index register has a "busy " rung (YINDBUSY) in its XREGLIST item; this contains a logical value which, when TRUE, indicates that this register is currently in use by ADRCALC (or others) as an X-reg in the midst of having an expression evaluated in it, and must not be disturbed by other processes. While it is busy, it will not be re-allocated by GETINDEX for other use. Consider, for instance, the quantity A(I+A(J+A(K))). In dealing with it, ADRCALC first wants to compute the subscript I+A(J+A(K)) in X1. It sets X1 busy and calls GTACUINT to evaluate that subscript in X1. While doing so, GTACUINT accesses A(J+A(K)), causing ADRCALC to be entered (recursively) for the purpose of setting up a register for the subscript J+A(K). X1 contains partial results for the evaluation of the outer subscript, so must not be reused. Another register, X3, is picked and "busied" for evaluation of the current script. Similarly, a third register, X5, is used for evaluation of the innermost subscript, K. Each register is "freed" when, its value being completely formed, it is used to access a subscripted quantity: X5 is freed first when A(K) is accessed; then X3 when A(J+A(K)) is accessed; finally X1 when A(I+A(J+A(K))) is accessed. The limitation of this scheme is reached when ADRCALC needs one more register and none is free. This problem is avoided through the use of the "number of busy regs" count cell, BUSYXCNT, maintained by BUSYX and FREEX. When ADRCALC needs to allocate a register, it examines this cell; if it is about to use the last free register, it does not try to gen subscript evaluation code in-line; instead, it pretends that the value is not computable in an index register (just as if the NEVALXFG trait were present) and uses the Retroactive Subscript mechanism to move the evaluation code back to a place where sufficient registers are available. The subroutines BUSYX and FREEX simply set the YINDBUSY rung TRUE or FALSE and modify BUSYXCNT.

2.14.4 Retroactive Subscript Gen SCRPTGEN)

SCRPTGEN provides the other end of ADRCALC's retroactive subscript mechanism; its function is to generate, for each expression/temp pair on n the SCRPTLST, code to evaluate the expression and store its value in the temp (an Integer Private Temp). Such code is placed up ahead of the current statement code on the Code List. See the description of ADRCALC (2.14.3.1) for a detailed discussion of the conditions under which script code needs to be evaluated retroactively; generally speaking, it happens when subscript expressions are too complicated to be evaluated directly in an index register by in-line code, or when subscript nesting gets too deep. SCRPTGEN is called (from STATEXIT)

after the statement driver has generated the main statement code; it delays that code by copying it to the Delayed Code List, gens subscript evaluation code on the Code List, then brings the statement code back to the Code List. Consider the statement X = A(2+F(X)); ADRCALC notes that the subscript 2+F(X) is complicated and puts it on the Script List along with a Private Temp pointer (say, T). The statement code produced is simply X = A(T). Later on, when SCRPTGEN is called, the code for T = 2+F(X) is generated ahead of the other code:

```
T = 2+F(X)

X = A(T)
```

Now suppose the subscript is one stage more complicated: X = A(2+F(A(3+F(X)))). Assignment Gen produces X = A(T1) and places T1 with 2+F(A(3+F(X))) on the Script List. When SCRPTGEN attempts to evaluate that expression, it runs into the same problem with the inner subscript 3+F(X). It generates T1 = 2+F(A(T2)) and puts T2 with 3+F(X) on the Script List. Having finished with the first "generation" of script code, SCRPTGEN proceeds to deal with the "children" thereof, in this case producing code for T2 = 3+F(X); this code is placed ahead of that for the first generation, resulting in the final statement code:

```
T2 = 3+F(X)

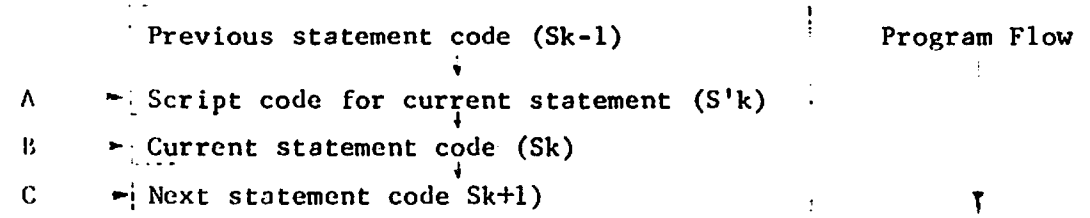
T1 = 2+F(A(T2))

X = A(T1)
```

No limitation is placed on the number of subscript generations. At each generation level, the Script List contents are copied to the Work List (emptying SCRPTLST) and this level's subscript evaluation code is generated (from information on the Work List) while the next level's script information is accumulated on the Script List.

We come now to SCRPTGEN's major problem: that of manipulating the register context. Down inside a nest of DO-loops, we are remembering across statement boundaries the contents of certain registers: X-registers containing loop-independent (c.f. 3.3) subscript values, and DO-registers holding DO-loop control variables. The totality of information pertaining to the contents of these registers is kept in the X-register List (XREGLIST) and DO-register List (DREGLIST) in a manner described in 2.17. We will call that information the register context. At any given time, code generation routines (ADRCALC in particular) may be using and/or modifying the register context in generating subscript references and other code involving X- and DO-registers; thus, the register context must at all times accurately represent conditions established by immediately preceding code. This raises a problem for SCRPTGEN and all other routines which generate code out of execution sequence: when code is moved around after it is generated, the register context must be manipulated to maintain the validity of assumptions made by the code generator concerning register contents. For convenience in discussing the problem as it relates to

SCRPTGEN, consider the following diagram of general statement flow with the labels A, B, and C applied at critical points:



Just after completing the generation of the previous statement, Sk-1, the register context reflects conditions holding at point A; based on this we generate the current statement code, Sk. When this is done, register context indicates conditions at C. Now if there is no retroactive subscript code to be generated (S'k), then everything is fine: we may proceed with the generation of the next statement without delay. But, if there is such code, SCRPTGEN wants to put code in the A-to-B region and encounters three troubles:

(1) Register context reflects status at C, not at A; thus its a prior assumption about initial register contents is not satisfied.

(2) Even if that were not the case, the S'k code will modify the register contents; they will not be the same as they were at A, thus invalidating assumptions made at the beginning of main statement code (Sk) generation.

(3) At the end of the SCRPTGEN process, the register context corresponds to point B, instead of C, as it should for proper generation Sk+1.

Item (1) above is actually a repeating trouble for SCRPTGEN, as it occurs anew for each level of subscript code generated; what we need is for the register contents to be as they were at A at the beginning of each such level, not just the first.

These troubles are resolved as follows. First, before generating any subscript code, SCRPTGEN saves away the current (point C) register context and restores it as it was at point A, before the statement code was genned (see below for details of this); we now have the correct context for generation of the first layer of script code. A copy of this context is placed on the Begin List (equated internally to TERMLIST); after generating each layer of script code, SCRPTLST compares the then current register context with this saved version and generates code to put changed registers back the way they were at point A. This takes care of troubles (1) and (2); (3) is resolved when, at the end of the SCRPTGEN process, we restore the point-C register context which we saved at the beginning.

A more detailed description of SCRPTGEN follows. First, we test SCRPTLST; if it is empty, then there is nothing to do, so we exit. If it is not empty, we proceed as outlined above, starting with a call on REVXOML (Reverse X/O status) which "swaps" XREGLIST with TXREGLST and OREGLIST with TOREGLST (actually, it is a swap of the list parameters, not the contents; it amounts to the same thing, but much faster). In this way, we save the point-C context for later and bring in the point-A context saved by a call on REMXOML at the beginning of the statement generation (in STGNWO). We now make a copy of this context on the Begin List, move the statement code from the Code List to the Delayed Code List, and RSV the Save List (SAVELIST). The latter will be used to accumulate X-reg save items, each consisting of an X-reg pointer and an integer temp pointer; this is part of the mechanism for restoring clobbered XOregs after each layer of script code (see below).

We are now prepeared to generate the first (outermost) layer of subscript code (the Code List is empty at this point); we CAE the Script List contents to a file on the Work List and, for each expression temp pair therein, call GTACUINT to evaluate the expression and gen code to store its value in the temp. Each call on GTACUINT may result in retroactive subscript code for the next layer; it will appear on the Script List. After all of the current layer's subscript code has been thus produced, we generate code to restore the point-A register context; this is done by a sweep through the XREGLIST items, then one through OREGLIST:

- For each X-reg item, CHECKXML is ca led upon to renew the XREGLIST item (i.e., put it back the way it is on the Begin List) and check to see if it is a loop-independent quantity (LOOPINDP trait) which got changed (the pre- renewal XREGLIST item differed from the Begin List version); if so, the X-reg has to be saved (at the beginning) and restored (now), so we put the X-reg pointer and an integer temp pointer on the Save List (if it does not already appear there) and gen a load of the X-reg from the temp.

- For each DO-reg, we call upon CHECKOML to see if it is a currently dedicated DO-loop control register which got clobbered by this layer's script code (i.e., if its YDOCLOB rung was FALSE in the Begin List but TRUE in the OREGLIST); if it was, we call RESTORDO to restore its contents (and the corresponding context).

This completes the generation of one level of subscript code; it is now on the Code List. We combine it with outer subscript code by copying Delayed Code List onto Code List. If there is another layer of subscript code to he generated (determined by seeing if the Script List is now empty), we copy this combined code back to the Delayed Code List, emptying the Code List, and proceed as before to generate the next layer. When there is no more subscript code to be genned, we RLS the file of saved register context on the Begin List and see if the Save List is non-empty. If it is, there are X-reg stores to be generated, so we again delay the subscript code, gen a STW of each X-reg to be saved in its corresponding temp, then copy the delayed subscript code back. In either case, we finally copy the delayed statement code back to the Code List, call REVXOML to bring back the saved point-C register context, and exit.

Thus, the overall effect SCRPTGEN has is to place the following code ahead of the CODELIST contents:

1 Store Some X-regs (the ones that will be clobbered by (2a)... (Na) below) into temps:

2a evaulace youngest Retroactive Subscripts; store them into temps;

2b Restore clobbered X-regs from temps (saved at (1) above) and DO-regs from control variables;

.

.

.

(Na) evaluate and store eldest Retroactive Subscripts;

(Nb) restore X- and DO-regs.

2.14.5 Delayed Code

The code generation POPs (BIF, BIN, etc.) build code on the Code List rathen than directly on the Output List primarily so that sequences of code can be generated and then be moved around relative to each other. The usual way of doing this is to RSV the Code List, generate some code, then copy the Code List file to the Delayed Code List (DCODLIST); now the Code List is available for accumulating new code; the code genned earlier may then be copied back to the Code List, placing that code physica ly after the code genned later. This kind of thing is done e by many of the statement drivers. For instance, DO-loops, REPEAT loops, and the implied DO-loops in I/O lists are handled by generating both top-of-loop and bottom-of-loop code in response to the loop-top (DO, etc.) statement. The Top code is generated first on the Code List, the Code List is RSVed, then the Bottom code is generated and copied from the Code List to the Delayed Code List. When the "End of Loop" interface item comes along, we copy the file of delayed code from DCODLIST back to CODELIST. Thus, at any given time, there is on DCODLIST a file of code for each level of currently unterminated DO-type loops. Such files of code are usually manipulated on a first-in-last- out basis since that usually is the order in which they are needed, but occassionally it gets trickier, as in "End of Illegally Nested loop" (see 3.17.2) where we unpile n files of delayed loop-end code, save one file (removing it), then stack the n files back on.

An important consideration to be made when delaying code is the effect such manipulation will have on :Retroactive Subscript: evaluation code (See 2.14.3 and 2.14.4 for a general discussion of the subscripting problem). Usually as mentioned in 2.14.1, statement code is genned by first calling the statement generator, then calling SCRPTGEN to put additional subscript evaluation code up ahead of the items already on the Code List. If the statement driver produces more than one file of code with the intent of moving them around to some sequence other than that in which they were genned, care must be taken to assure that retroactive subscript code gets genned in appropriate places. The basic rule is: whenever you are going to generate one file of code, you start with a RSV of CODELIST and end with a JRS to SCRPTGEN; any file of code produced in this way is safe to dealy. For example, each REPEAT FOR triple causes three files of code to be produced: one for the loop-top code and two (one for Step and one for test) to be delayed as loop-bottom code. Even though the latter two files get lumped together finally, they are generated as distinct files so that Test's subscript code will be placed at the Test entry point rather than up ahead of Step, where it would not be executed the first time through the loop. In conformance with the rule, REPEAT Triple Gen (TRIPGEN) executes three RSV CODELISTs and three JRS SCRPTGENs.

Code is also occassionally delayed on three other lists, all associated with the Debug interface: Source Line List (SRCLNLST), Entry List (ENTRYLST), and Label List (LABELLST). The Debug Table Gen routine DBTBLGEN may copy the contents of these lists onto CODELIST. See also 3.19.

2.15 Assemble Code ASMCODE)

ASMCODE reads code (in the form of Pass2/Pass 3 interface items from the Code List and places it on the Output List. ASMCODE behaves much like the first pass of a simple assembler, in that it now knows the program location in which each instruction is to appear and can therefore define labels explicitly. Most of the code passes through ASMCODE without modification; the exceptions are as follows:

- The operands of (address pointers) of normal instructions are checked: if we encounter a label (user label, not a created label) which undefined, we discard the given instruction and substitute BAL,6 9UNDEFLB;

- If an instruction's operand (address) is a pointer to one of the constant lists, we set it Constant Used rung;

- In response to a Build Library Call item, ASMCODE places the referenced library routine name on the Ref List so that Pass 3 will generate an external REF to it;

- A Define Label item causes the current location counter (LOCCOUNT) value to be associated with the label or symbol:

  a) If the operand is a Symbol Table pointer, (i.e., we are defining the location of a subprogram entry point), LOCCOUNT is stored into the YALOCPTR rung of the Symbol Table item;

  b) For labels, LOCCOUNTis stored into the LBDEFWD runf for the label, if that rung's previous value was zero; in this manner, the first definition of a multiply defined label is retained;

  c) For a created label, in addition to defining its location as above, we append (in its CRELBWD rung) the Created Label Count and increment it so that Created Label numbers will appear in program order.

2.16
Naming Conventions for Expression Evaluation Generation Subroutines

There is considerable consistency in the naming fo the routines that generate code to evaluate arithmetic expressions. There are 58 different entry points * to the expression evaluator, providing minor variations as to where the result is to be found (in memory, in an accumulator, etc.), what mode the result is to be, and whether or not the result can be computed with the wrong sign.

The names are all eight characters in length. Of these, the first two are always the letters "GT", standing for "GuaranTee" (comments in the listing almost always abbreviate this word as "GRNTEE"). The next two letters tell where the result goes; the next letter is for the sign convention, and the last three for the mode convention. The following table indicates the possibilities.

Guarantee	Where result goes	Sign convention	Mode convention
Chars 1 & 2	Chars 3 & 4	Char 5	Chars 6, 7 & 8
GT	AC Accumulator	U Unsigned	BYM By Mode
	AD Addressable	S Signed**	OWN Own Mode
	BO Both the above	X Signed, with sign to Answer Flag	KNW Known Mode
	CC Condition codes		REM Remember Mode
			FGT Forget Mode
			WIM WI Mode
			INT Integer
			REA Real
			DBL Double
			CPX Complex
			LOG Logical
			OPR Operand
			CPO Complex Operand

* Of which twenty are not referenced.

** With sign appended to the bottom of the SIGNLIST.

Entrances with By Mode (BYM) in their names assume that the desired mode has already been set up on the MODELIST and that all the elements should be converted to that mode. These entrances are used mainly from within the expression generator. For example, for the denominator in division, the entry point used is Grntee Signed Addressable By Mode (GTADSBYM), meaning: get the result in memory; allow the wrong sing, and convert the modes of the elements individually to the mode on the MODELIST.

The entrances with Own Mode IOWN) in their names mean: set up the highest mode of the expression onto the MODELIST; generate the expression in that mode, and then remove the mode from the MODELIST. These entrances are used forr subprogram arguments and similar items. Among these names, Grntee Addressable Own Mode (GTAUOWN) is special and does more than its name implies. In fact it handles all the work of arguments for standard subprogram calls, such as allowing statement labels and subprogram names. This entrance also allows logical expressions, and is the only ostensibly arithmetic entrance that does so.

The entrances labeled Remember Mode (REM) are just like the Own Mode entrances except that tey do not remove the mode from the MODELIST at the end.

The entrances labeled Known Mode (KNW) cause an expression to be evaluated in its own mode, then converted to the mode on the bottom of the MODELIST. This is quite similar to the By Mode entrances in that the mode becomes whatever the MODELIST indicates. The difference is that in By Mode the conversion is performed individually for each of the elements, whereas in Known Mode the conversion is performed only once, at the end. The Know Mode entrances are quite popular for evaluating arguments to intrinsic functions, and are also used in the DO statement.

Entrances marked Forget Mode (FGT) are just like Know Mode except that they remove the mode from the MODELIST at the end.

Entrances marked Wl Mode (WlM) take the mode from the item of the WORKLIST, append this mode to the MODELIST then evaluate the expression By Mode, then throw away the mode. These entry points are used in handling intrinsic functions.

Then there are entrances marked with the individual modes themselves by name: integer (INT), real (REA), double (DBL), complex (CXP), and logical (LOG). Of these, only integer and logical are actually used. The others had some use in the 9300 intrinsic functions, but no more. The entry point for Grntee Logical Expression Addressable (GTADULOG) is consistent with the others, but Grntee Logical Expression In AC is in fact called Evaluate Logical Expression (EVALOGEX).

Entrances with the word Operand (OPR) or Complex Operand (CPO) in their names do strange things that are useful only within the bowels of the expression generator. They are discussed in 3.10.2.2.

It is almost correct to say that if a name in Pass 2 begins with the letters GT it is an expression evaluator. But not quite. Sometimes the GT is the .GT. (Greater Than) operator. There are six names in this category: GTJUMPF (.GT. Gen Jump if FALSE), GTJUMPT (.GT. Gen Jump if TRUE), GTPLEX (.GT. Plex), GTOJUMPF (.GT. Zero Gen Jump if False), GTOJUMPT (.GT. Zero Gen Jump if TRUE), and GTOPLEX (.GT. Zero Plex).

There are a few more "GT" names where the GT means Guarantee, but which do not otherwise follow the format in the table. These include GTEAX, meaning Grntee Effective Address In Index, and its companions GTEAX5 and GTEAX7.

2.17 Register Context Handling

This section deals with index registers (X-regs) and DO-loop control registers (DO-regs): how they are allocated and how the compiler remembers what's in them. The X-regs are registers 1,3,5, and 7; the DO-regs are 2,4, and 6.

2.17.1 X-reg Context Storage, XREGLIST

Context for the X-regs is remembered on the X-reg List (XREGLIST). This list (like UNITLIST, for example) is a static list; that is, its size never changes and it is assembled in place with certain fixed data and cells which information may be placed by the compiler. There is one six-word item in XREGLIST for each of the four X-regs, and these items appear in the order of corresponding X-regs 1,3,5,7. The six rungs of each item are as follows:

Number	Name	Contents/Use
0	YINDWHAT	An expression pointer representing the known contents of the X-reg; it is zero if the X-reg is "clobbered" (not known).
1	YINDADND	Constant addend: the difference between the true X-reg contents and the above expression value.
2	YINDBUSY	the "busy X-reg" trigger (see 2.14.3.3).
3*	YINDEXNR	The X-reg number; i.e., 1,3,5, or 7.
4*	YINDEXX	The X-reg number shifted to the index field (bits 12-14).
5*	YINDEXR	The X-reg number shifted to the register field (bits 8-11).

The rungs whose numbers are starred above are fixed data; that is, they are assembled into the list's permanent storage area and not changed during compilations. An "X-reg pointer" is an XREGLIST pointer to the zeroth rung (YINDWHAT) of one of the four items; context data for a particular X-reg is accessed by addressing relative to that X-reg's pointer.

2.17.2 DO-reg Context Storage, DREGLIST

Context for the Do-regs is remembered on the DO-reg List (DREGLIST). This is also static list, and it contains one seven-word item for each of the three DO-regs. These items appear in the order of corresponding DO-regs 4,2, and 6. (This somewhat surprising ordering is chosen so that the first DO-reg picked (4) is that which is least likely to be clobbered by a library call; 6 is always clobbered). The seven rungs of these items are as follows:

Number	Name	Contents/Use
0	YDOCVAR	Symbol table pointer to loop's control variable; it is zero if the DP-reg is not currently dedicated to a loop
1	YDOADND	Constant addend: the difference between the true DO-reg contents and the control variable's value
2	YDOCLOB	The "DO-reg clobbered temporarily" trigger (see below)
3*	YDONR	The DO-reg number; i.e., 2, 4, or 6
4*	YDOX	The DO-reg number shifted to the index field (bits 12-14)
5*	YDOR	The DO-reg number shifted to the register field (bits 8-11)
6	YDOLEVEL	Pointer to DOLIST Used as an indication of nesting depth

The rungs whose numbers are starred are fixed data. Note the parallels in organization of XREGLIST and DREGLIST: in either case, the register's contents is the value of the expression pointed to by rung 0 (YINDWHAT/YDOCVAR) plus the integer constant in rung 1 (YINDADND/YDOADND); in either case, the fixed data representing the register's number appears in the same rungs (YINDEXNR/YDONR, YINDEXX/YDOX, and YINDEXR/YDOR). ADRCALC accesses context data for X-regs and DO-regs without needing to know the difference. A "DO-reg pointer" is a DREGLIST pointer to rung) (YDOCVAR) of one of the three DO-reg items.

The YDOCLOB rung contains a trigger which remembers whether or not the DO-reg currently contains the value defined by YDOCVAR and YDOADND. Unlike X-regs, a DO-reg may be "half remembered"; that is, it may still be dedicated to a loop (as indicated by the nonzero control variable pointer in YDOCVAR) but the register's value may have been temporarily clobbered by a subroutine call or the like; in this case, the value may be restored (via RESTORDO). See 2.17.5 for further details on this.

Since there is no limit on DO-loop nesting depth, and there is a limit on available DO-regs, there is a mechanism which enables us to save (in dynamic storage) the DO-reg context representing the current use of a particular DO-reg, re-dedicate that register to a new loop, and (when that loop ends) restore the saved context. When a DO-loop is being built (see DO Statement Gen, 3.38.2), a four-word file is placed on the DOLIST: enough information to restore the non-fixed rungs of a DREGLIST item. They are as follows:

1. the YDOCVAR rung's contents

2. the YDOADND rung's contents

3. the YDOLEVEL rung's contents

4. a DO-reg pointer, identifying the particular item to be restored

This file is removed from the DOLIST (and the corresponding DREGLIST item is restored) by End of Loop Gen (see 3.38.3). Note that the remaining non-fixed context item YDOCLOB need not be saved in this manner as, at End-of-Loop time, the DO-reg's contents are known to have been clobbered by the inner loop's use of it. A pointer to this DOLIST file is placed in the DO-reg's YDOLEVEL rung for use by GETDOREG (2.17.4) as an indication of nesting level; that pointer is <u>not</u> used to access the saved information.

2.17.3 Remembering Register Context En Masse, TXREGLST, TDREGLST

TRXRELST (The Temporary X-reg List) and TDREGLST (Temporary DO-reg List are fixed-length buffers (not actual lists, despite their misleading names) that are used together to contain one additional copy of the entire register context. They are used by Script Gen (via the subroutines described below) to save and recall the register context as required for the code rearrangement it performs (see 2.14.4).

There are two machine language subroutines for moving data to and from these lists:

- Remember Current X-reg/DO-reg Status (REMXDML(puts a copy of all of the XREGLIST contents on TXREGLST, and all of the DREGLIST contents on TDREGLST; in this way it saves the entire register context.

- Reverse X-reg/DO-reg Status (REVXDML) swaps the current and saved copies of register context by effectively exchanging the contents of XREGLIST with TXREGLST, and those of DREGLIST with TDREGLST; it is not the list contents but their location parameters (Bases, Starts, Tops, and Bottoms) that are exchanged.

2.17.4 Register Allocation

To get an arbitrary X-reg, ADRCALC calls GETINDEX, which returns (in WO) a pointer to the next available non-busy X-reg. It samples the XREGLIST items cyclically (1, 3, 5, 7, 1, 3, ...) and stops on the first one whose YINDBUSY trigger is FALSE. It keeps track of the last X-reg allocated by saving its pointer in the cell NEXTINDX, which is advanced to a new X-reg item by means of the PNI POP, restoring it to X-reg List Top once the Bottom has been passed. The process is initialized by CLOBINDX which frees all the X-regs and sets NEXTINDX to X-reg List Top. NEXTINDX is also modified by GTEAX (see 2.14.2) to force the selection (by GETINDEX) of a particular X-reg.

DO-reg allocation is performed by GETDOREG which returns a pointer to the first (in sequence 4,2,6) undedicated DO-reg available, or the one dedicated to the outermost loop, if they are all in use. It does this by making a sweep through the DO-reg List items looking for the one with the smallest YDOLEVEL rung (outermost loop); if it comes across one with a zero level rung, indicating an unused DO-reg, the search stops and that DO-reg is selected. Initialization is done by INITDREG which zeros the YDOCVAR and YDOLEVEL rungs of all DO-reg List items.

2.17.5 Register Clobbering, DO-reg Restoring

There are four subroutines which may be called to forget the contents of groups of registers: one may clobber the contents of all DO-regs, all X-regs, the X-regs containing only loop-dependent quantities, or all of the above.

1. CLOBINDX clobbers all XOregs by (a) setting all YINDWHAT rungs to zero and YINDBUSY rungs FALSE, (b) setting the Busy Index Count (BUSYXCNT) to zero, and (c) setting Clobber X Trigger (CLOBXTRG) FALSE so that X-regs will not be clobbered again, unnecessarily, until there is reason to.

2. CLOBDEPX (Clobber Dependent X-regs) branches to CLOBINDX if we are not currently in a loop (DOLIST is empty); if we are in a loop, it performs like CLOBINDX, except that it zeros only those YINDWHAT rungs whose expressions do not have the Loop_Independent trait (LOOPINDP); it frees all the X-regs, as above.

3. CLOBDML (Clobber DO-reg, Machine Language) simply sets all the YDOCLOB rungs TRUE, since these registers are only to be "half-forgotten" (they are still dedicated to their respective loops).

4. CLOBALL (Clobber All registers) calls CLOBDML then falls into CLOBINDX.

To restore the contents of a "half-forgotten" (i.e., clobbered but still dedicated) DO-reg, Restore DO-reg (RESTORDO) is called. You give it a DO-reg pointer in W0 and, if RESTORDO finds that that DO-reg's Clobbered Trigger is indeed set, it generates code to load the control variable into the register and add the addend to it (if nonzero, of course), and it resets the Clobbered Trigger.

2.18 Register Usage in Object Code

Fifteen of the sixteen hardware registers are divided up into three classes according to their use in object code. Register zero is not explicitly used by the object code since it is reserved for a stack pointer doubleword used by re-entrant library routines.

2.18.1 X-regs

Registers 1, 3, 5, and 7 are designated X-regs (Index registers) and are used to handle subscripting. Only the odd hardware index registers are included so that we can evaluate simple integer subscript expressions in them. The nature of the hardware integer multiply (MW, MI) and divide (DW) instructions forces us to use odd registers for FORTRAN expressions involving multiplies and divides. Here is an example of X-reg usage:

```
A(J+4) = A(J)/B(M*J-N)+C(K)

         LW,1 J

         LW,8 A-1,1

         LW,3 M
```

```
MW,3 J

SW,3 N

FDS,8     B-1,3

LW,5 K

FAS,8     C-1,5

STW,8     A+3,1
```

Note in the example that the use of X-regs is more "local" than that of the accumulator (register 8 here); that is, a quantity evaluated in an X--reg is used immediately and no longer needed - but the value in the accumulator is being worked on throughout the statement and must not be modified by the occasional subscript evaluation code. Quantities computed in X-regs are remembered (see 2.17 and 2.14.3) as long as we know that they are still there, so that they may be re-used, as in the last instruction above.

2.18.2 DO-regs

Registers 2, 4, and 6 are designated DO-regs (DO-loop control registers) and are used to hold the control variables (possibly offset by a constant amount) of certain DO-loops (see also 3.38.2). At the beginning of such a loop, one of these registers is assigned to the loop and loaded with the control variable; within the loop, the DO-reg might be temporarily re-used (for an inner loop, if nesting is deep enough) or clobbered (by a subroutine call, etc.), but its value is easily restorable, requiring at most a load (LW) and an add (AI). Since there are no multiplies or divides involved, there are no parity restrictions. Hardware index registers were chosen so that they could be used for subscripting where the subscript is a DO-loop control variable plus some constant. DO-regs, unlike X-regs, are subject to being clobbered by library routines; register 6 is used as the link to all of these subroutines, and 2 and/or 4 are modified by particular ones. The compiler has knowledge of which registers are clobbered by which library routines, and uses this information to set the Clobbered Trigger for those DO-regs in DREGLIST (see 2.17.2) when the BLC POP is used to generate a library call.

2.18.3 Accumulators

Registers 8 through 15 are used to hold values of any of the FORTRAN data types. Associated with each type there is a register or group of registers in this range called the accumulator for that type. Arithmetic operations are performed in the accumulator (see the code example in 2.18.1); arguments to library subroutines are computed in the accumulator. Some subroutines require the passage of two arguments; in such cases, there is a secondary accumulator associated with the type. Due to characteristics of the hardware integer multiply (MW, MI) and divide (DW) instructions, integer arithmetic must be done in an odd register and, due to a historical problem with the hardware instruction used for real multiply (FMS), real arithmetic is done in an even register. Double precision arithmetic requires, of course, an even/odd pair. The following table gives the actual register assignments:

Type	Primary Accumulator	Secondary Accumulator
Integer	9	11
Real	8	10
Double Precision	8-9	10-11
Complex	8-9	10-11
Double Complex	8-11	12-15
Logical	9	(none needed)

2.19 Temps

Temps are temporary storage cells that are dreamed up by the compiler to hold intermediate results during the object program's execution. These are in addition to the user's variables, which have names and which the user thinks up himself. The temps are allocated in the Local Variables control section and come after the user's local variables and dummies.

2.19.1 TEMPs, PTMPs, and RTMPs

On the object code listing, temps are referred to by a name consisting of an integer followed by one of the following three words:

TEMP

PTMP

RTMP

These names refer, respectively, to Ordinary Temps, Private Temps, and Real-Time Temps. Ordinary temps are used for intermediate results in the calculation of expressions and are generally reused on a statement-by-statement basis. (Exceptions to the reuse are mentioned in 2.19.5.) Private temps have a significance over several statements and hence are not reused, i.e., they are not shared. They are used for such things as DO-loop parameters and return-links for subprograms. Real-time temps have to do with recursion and reentrancy processing when the program is compiled under the "RT" option.

2.19.2 Temp Pointers

Temp pointers look very much like list pointers, except that they do not happen to point to any list. They have an ID field, a mode, and a displacement, the latter being a displacement from the start of that particular class of temps. A displacement of zero is listed as 1TEMP (or 1PTMP or 1RTMP), so in a sense the displacements are all off by one.

There are three different possible ID fields, corresponding to the three different kinds of temps. In the spectrum of ID field numbers (see 1.12) these are called Fake List Pointers. All these pointers have the traits Addressable (ADDR), Signed Addressable (SADR), and Test Addressable (TADR). They are defined in LISTDEFS.

2.19.3 Private Temps PTMPs

Private temps are usually generated in integer mode, but sometimes real or double precision private temps are made. If a double precision private temp is generated, then the block of private temps as a whole must start on a doubleword boundary. If the Private Temp Counter (PTEMPCNT) is odd when we want a Double Precision temp, we add 1 to it and just waste a word.

The subroutines concerned with making Private Temps are: Private Integer Temp Maker (PITEMPMK), Private Temp By Mode Maker (PTEMPMMK), Private Temp Maker (PTEMPMK), Store In Private Integer Temp (STPITEMP), Store In Private Temp By Mode (STPTEMPM), and Grntee Addressable Via PTMP, Known Mode (GTADPKNW). All of these subroutines are located in PASS2F except PITEMPMK, which exists in both PASS2F and PASSOF; the version in PASSOF is defined externally; the one in PASS2F is not. (The version in PASS2F could be eliminated and the external version used, but it uses only two POPs, so the change is not worth the trouble.) The reason for having a copy in PASSOF is that Pass 1 (in STASCAN) creates some private temps of its own certain circumstances.

There are essentially seven different situations in which PTMPs are used, and they are as follows:

A. Subprogram Links These are the RETURN locations supplied by the BAL,15 instruction used in standard calling sequences for SUBROUTINE, FUNCTION, and ENTRY statements as well as in Statement Function Definition statements. The RETURN sequence consists of an indirect branch through the resulting Private Temp. A less obvious link location is the one generated for a REPEAT FOR statement whose value list has two or more items in it. In this case, the code controlled by the loop is made into a subroutine and called by the loop driver. Here the link used is also Register 15.

B. DO Statement Parameters For certain kinds of DO statements, an iteration count may be stored in a private temp. This count will be integer, regardless of the mode of the induction variable. For other kinds of DO statements, the step and termination values (which are computed at the beginning of execution) are stored in private temps. These are kept in the mode of the induction variable.

C. Adjustable Dummy Array Parameters If a dummy arrray has adjustable dimensions, it is probable that its subscript multipliers are non-constant. Since these multipliers are used every time an element of the array is mentioned, we compute them at prologue time (the time of the SUBROUTINE, FUNCTION or ENTRY statement) instead of at each reference. A subscript multiplier is the upper bound minus the lower bound of a particular dimension, plus one. Multipliers are needed for all but the right-most dimension. In addition, the array offset is computed at prologue time and also stored in an integer private temp. The offset is the difference between the location of the first element of the array and the zeroth element (the one at which all subscript values are zero). These private temps are all created by Pass 1 (STASCAN). Pointers to them are kept on the ARRAYLST and the GROUPLST. (See also Arrays, 3.2.

D. Number of Elements of an Adjustable Array This is really an extension of C above, but it is a parameter so rarely needed that we do not generate it unless we know it is required. The only place an object program has to know explicitly the size of an array is when that array is used unsubscripted in an I/O list. Such usage means that the whole array is to be input or output, and SIOLUSA has to know how many elements to send. In the Symbol Table we keep a trait called IOLUSAFG (I/O List UnSubscripted Array FlaG) indicating whether an array has ever been used in this way. If an adjustable dummy array is ever used unsubscripted in an I/O list, we generate code at prologue time to compute the element count and store it in a private integer temp. This temp is created by Pass 2 since we do not know until we have scanned the whole program whether the array ever got used unsubscripted.

E. OUTPUT Statement The OUTPUT statement is translated by Pass 1 into one or more BCD WRITE statements and one or more FORMAT statements. If the complexity of the expression list is great enough, Pass 1 may have to reproduce more than one WRITE statement, and for each of these the unit number must be passed. The unit number may be an expression of some

complexity, and it would be silly to recompute it several times, so if the unit number is at all complicated it is (usually just a constant), Pass 1 generates an assignment statement of the unit number into a Private Temp, and does further references to the unit number from that temp.

F. CONNECT Statement For sheer bulk of Private Temp usage, the CONNECT statement wins the prize. Every CONNECT statement takes a block of 26 consecutive private temps, to be used as a Task Control Block (TCB). These 26 words must start in an even location, so we may have to waster a word before the block. This block of temps is created by making a Double Precision private temp (this assures an even location), then adding 24 to the Private Temp Counter (PTEMPCNT).

G. DEBUG Symbol Table Chain When the "DEBUG" option is specified, the compiler generates a symbol table much like that for the NAMELIST statement that makes the user's variable names usable at execution time. If the results of several different compilations are loaded, the Debug symbol tables of each are chained together, and the system needs a Double Precision Private Temp to do this chaining with. The address of this private tmep is passed to SDBINIT at execution time. (See also Debug Tables, 3.19.2.)

2.19.4 Ordinary Temps TEMPs)

Unlike private temps, ordinary temps are shared across several statements. Within one statement they are, for the most part, not shared, and each temp serves a single purpose. Thus the user can look at the object code for a single statement and say, "My, the compiler wasted a lot of temps." But usually across a whole program the compiler will not waste more than ten temps, and that is not so bad. (The reason that we do not reuse TEMPs within a statement is that it is hard to know when we are truly finished with a temp.)

TEMPs are generally used for intermediate results in expression calculation. They are also used to hold argument addresses for calling sequences when such argument addresses must be computed at execution time.

TEMPs are used for holding subscript values (index register contents) that must be saved and restored. They also hold the computed values of ugly subscripts that are generated retroactively at the beginning of statements.

Two statements make special and peculiar use of ordinary temps: the REPEAT FOR statement may use an integer temp to hold the sign of the step value, to remember whether the loop is running up or down; and the ENCODE and DECODE statements may, under perverse circumstances, store the value of the last parameter--the number of characters processed -in a temp.

The subroutines concerned with the creation of TEMPs are all in Pass 2. They are: Integer Temp Maker (ITEMPMK), Logical Temp Maker (LOGTEMPM), Temp By Mode

Maker (TEMPMMKR), Store in Temp By Mode (STEMPBYM), Store In Integer Temp (STORITMP), and Temp Maker (TEMPMKR).* All these routines are located in PASS2F.

The Temp makers are quite analogous to their counterparts among the Private Temp makers, but there is a difference in the way double-word Temps are handled. Although double-word Private Temps are comparatively rare, double-word Temps are quite common, and it is not unusual to see them in the same statement with single-word temps. When the Private Temp makers want to make a double-word temp and the Private Temp Counter is odd, they just add one to it and waste a Private Temp. But when the same situation arises in the Temp makers, we skip over a single-word temp, all right, but we make an effort to save it for later. The next time anybody wants a single-word temp in this statement, he will get the temp that was just skipped rather than advancing the Temp Counter (TEMPCNT) to the next available word. This is handled through a cell called Spare Temp (SPARTEMP) that is zero if there is no spare temp waiting to be used, and contains a temp pointer if there is a spare single-word temp available. Spare Temp (SPARTEMP) is zeroed at the beginning of generation of every statement.

* Four other subroutines exist, but they are never called: Real Temp Maker (RTEMPMK), Double Temp Maker (DBLTEMPM), Store In Real Temp (STRLTMP), and Store in Double Temp (STDBTMP).

2.19.5 Privacy In Ordinary Temps

Although it is true that TEMPs are shared among statements, it is not true that all TEMPs are shared among all statements. There are barriers, so to speak. For example, a statement function must not use the same temps as the expression that calls it, for one would clobber the other.

If we were going to allow unrestricted sharing of the TEMPs, we would simply zero the TEMPCNT at the beginning of every statement. Instead, there is a more complicated procedure involving three different cells, all concerned with counting temps. There is the Temp Count (TEMPCNT), plus the Initial Temp Count (ITEMPCNT) and the Maximum Temp Count (MAXTEMPC).

At the beginning of Pass 2, both ITEMPCNT and MAXTEMPC are set to zero. Then at the beginning of each statement TEMPCNT is set equal to ITEMPCNT. Every time TEMPCNT is advanced, MAXTEMPC gets updated in case TEMPCNT has gone beyond it.

There is a little subroutine called Protect Temps (PROTEMPS) that sets both TEMPCNT and ITEMPCNT equal to MAXTEMPC. Note that this is the reverse of what is going on above: here a high value is replacing these normally low cells, and they can never get lower than this again.

Every time PROTEMPS is called, it essentially draws a line through the TEMPs. If a statment needs to be isolated from both its predecessors and its followers, then PROTEMPS must be called at both the beginning and the end of the generation of that statement.

2.19.6 Which Comes First?

When, during Pass 3, space for the TEMPs and PTMPs is being allocated, the question arises, "Which is to come first: the TEMPs or the PTMPs?" There is the possibility that we can save one word in generated object storage by arranging TEMPs and PTMPs so that odd/even parities work out nicely. The decision is based on the answers to the following questions:

A. Is VSIZE odd, i.e., have there been an odd number of locations allocated already for local variables and dummies?

B. Do the TEMPs need to start in an even location, i.e., is TEMPEVTR (Temp Even Trigger) TRUE?

C. Do the PTMPs need to start in an even location, i.e., is PTMPEVTR (Private Temp Even Trigger) TRUE?

D. Have an odd number of TEMPs been allocated, i.e., is MAXTEMPC odd?

E. Have an odd number of PTMPs been allocated, i.e., is PTMPCNT odd?

All this logic is performed in Allocate Temps (ALTEMPS) located in the assembly PASS3F.

No logic is performed to decide where to put the Real-Time Temps because there is at most one RTMP, and it must come at the very end of the Local Variable Storage CSECT Part of the RTMP contains a count of the number of words that precede it in Local Variable Storage. (See also Real Time, 3.12.)

2.19.7 The Real-Time Temp RTMP)

There is only one Real-Time Temp, at most. It comes at the end of the whole Local Variable Storage and indicates the size of that region, including the TEMPs and PTMPs. The layout of Local Variable Storage as a whole is

User-defined Local Variables

Dummies

TEMPs and PTMPs

RTMP

Any or all of these classes may be empty. RTMP is interpreted as two halfwords. The left half is a logical sentinel indicating whether the subprogram is being reentered, and the right half is a constant telling the size of local variable storage.

2.20 Mode Conversion

Essentially, every element of an expression is evaluated in the mode of the pointer. What this usually means is that each element has its mode "promoted" to the highest mode in the expression. (This is different from what many compilers do with mixed-mode expressions: they evaluate each operation in the higher mode of the two operands, but they do not allow the mode of an element somewhere else in the expression to have influence.)

I said "essentially" at the beginning of the previous paragraph, because this is not always true. In a complex or komplex expression, the non-complex elements are evaluated in the highest appropriate non-complex mode. This eliminates redundant computations on things whose imaginary parts are zero.

Sometimes the mode conversion is done in compiled code (calling the appropriate mode conversion library subroutine, or perhaps doing the conversion in-line), and in the case of constants the conversion is often done at compile time.

2.20.1 The MODELIST

In order to implement the mode conversion rule, Pass 2 has a Mode List (MODELIST), a push-down stack onto which it puts the expression pointer it is about to evaluate (i.e., generate evaluation code for). All of this pointer is ignored except the mode field. The MODELIST is set up at entry to any of the expression evaluators except those with By Mode or Known Mode (BYM or KNW) in its name. (These latter operate with the mode that is already there.)

2.20.2 Mode Conversion Gen (MODCVGEN)

Generation of the actual conversion code is done by MODCVGEN. This routine expects two mode indicators--the "from" indicator in W0 and the "to" indicator on the bottom of the MODELIST. The indicator in W0 is discarded. These two modes are combined into a single index that is used to address a 25-entry table that is XEQed. This is done by multiplying the WORKLIST mode by 5 and adding in the MODELIST mode, then subtracting six from the result. (The six may be thought of as the offset of the array.) The code generated for the twenty-five cases is shown in the following table.

	to I	to R	to D	to C	to K
from I		BAL,6 9ITOR	BAL,6 9ITOD	BAL,6 9ITOR LI,9 0	BAL,6 9ITOD LD,10 =0.0D0
from R	BAL,6 9RTOI		LI,9 0	LI,9 0	LI,9 0 LD,10 =0.0D0
from D	BAL,6 9DTOI	BAL,6 9DTOR		BAL,6 9DTOR LI,9 0	LD,10 =0.0D0
from C	BAL,6 9RTOI		LI,9 0		LI,11 0 LW,10 9 LI,9 0
from K	BAL,6 9DTOI	BAL,6 9DTOR		BAL,6 9KTOC	

2.20.3 Constant Mode Conversion

At many places in expression evaluation generation in Pass 2 we call either Correct Constant Mode (CORCONM) or Correct Constant Sign And Mode (CORCONSM). These both expect an expression pointer in WD and, if this is a pointer to one of the constant lists, converts it to a pointer to another constant. CORCONSM merely JRSs to Correct Constant Sign (CORCONS) and then falls into CORCONM, which, like Mode Conversion Gen (MODCVGEN, 2.22.2) indexes into a twenty- five word table. The table entries are XEQed, and most entries are a JRS to a routine named CONxTOy (Constant Conversion from mode x to mode y). The twenty subroutines so named are located in PASSOF, where they are also used by the DATA statement in Pass 1.

2.21 Pass 2/Pass 3 Interface

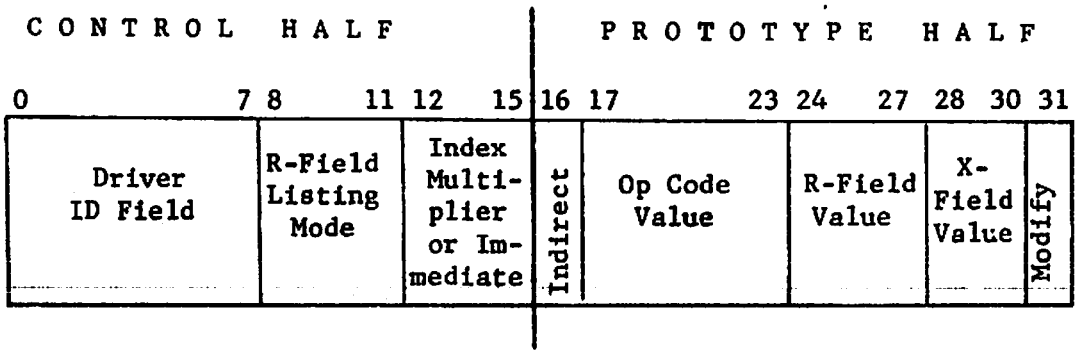
The interface between passes 2 and 3 is much the same as that described in Table 2.11.1 for the Pass 1/Pass 2 interface, except that the intermediate scratch file is X2 instead of X1. ASMCODE (Assemble Code) in Pass 2 leaves output on the Output List; it may, from time to time, be dumped to the scratch file. Pass 3 will bring it to the Input List and access it one-word-at-a-time from there via NXINWORD (Next Input Word).

The interface items are fewer in number and simpler in structure than those of the Pass 1/Pass 2 interface. The first word of an item is a pseudo-pointer; i.e., its type code appears in the ID field (first byte) of that word. The rest of the first word and subsequent words specify additional information used by the item generator. The most common item is OPCD, generated by Pass 2's BIF POP: its first word contains the op code, register field, index field, and indirect address bit of a Sigma hardware instruction, as well as information telling Pass 3 how to list it; the second word is the Address Pointer associated with the instruction - it tells Pass 3 how to derive the actual address to be placed in the instruction, and how it should be listed. Most of the interface items have this same structure.

The table given below lists the interface items in terms of symbolic codes only; the actual numerical code values are subject to change, as they are defined (in LISTDEFS) immediately following list and plex numbers. The names given in the table are the same as the names defined in LISTDEFS with their leading "P23" omitted. The statement drivers corresponding to the interface items (in OUTPROG) are preceded by comments giving the details thereof. In the table, a size indication of "V" represents a variable-length item.

Name		Number of Words	Statement type
HABS		V	Hollerith multiple-word absolute
OPCD		2	Op code (normal instruction)
IMMOP		2	Immediate op code (LI, AI, etc.)
BOPC		2	Branch op code (BG, BGE, etc.)
BOPCZ		2	Branch with Z op code (BGZ, BGEZ, etc.)
OPCS		2	Calling sequence op code (INTG, SNGL, etc.)
DEF	2		Label definition
ADND		1	Addend value
SOURC		V	Source line string
END	1		End of program
BLC	1		Build library call
FABS		2	Full word absolute instruction (LCI, SLS, etc.)
NABS		2	Numeric (hex) absolute data (DATA)
DBGEN		2	Debug GEN (GEN,1,14,17)
CNL	1		Comment next line
PLEXL		V	Plex list
DTORG		2	DATA origin
DCON		2	DATA constant
RPROG		1	Return to program origin

The first word of a typical interface item is structured as follows:



The first fifteen bits of a normal Sigma instruction (all except the address field) appear in the right half of the interface word. The left half (and bit 31) contain information used by Passes 2 and 3 for various purposes. The address of the instruction is represented by a pointer in the second word of the item, possibly with the help of an absolute Addend value represented in a preceding ADND interface item. This word's control fields are interpreted as follows:

Driver ID field — tells which Pass 2/Pass 3 interface item this is.

R-Field List Mode — tells OUTPROG how to list its R-field.

Index Mult/Immediate — tells ADRCALC what kind of indexing (word or doubleword) the instruction does; it also tells Pass 3 whether or not the instruction has immediate addressing.

Modify — tells Pass 2 if the instruction is a memory-modifying instruction, so that it will generate a Data Check-in if the DEBUG option is given.

This structure applies to the OPCD, IMMOP, BOPC, BOPCZ, and OPCS interface item3

2.22 Output Program (OUTPROG)

OUTPROG is what Pass 3 is all about. It is the only part of Pass 3 that can legitimately be called a pass, for it processes the file of generated object code output by Pass 2. About 75 per cent of the code in OUTPROG is concerned with producing the object listing, and much of this code is skipped when the LO option is not requested.

Pass 2 outputs nineteen different indicators that OUTPROG understands. These are as follows:

Hollerith Multiple-Word ABS

Normal OP Code

Immediate OP Code

Branch Op Code

Branch/Z Op Code

Calling Sequence Op Code

Label Definition

Addend

Source Line

End Of Program

Library Call (BAL,6 Inst)

Full-Word ABS Instruction

Numeric ABS

Debug Gen (GEN,1,14,17)

Comment Next Line

Load Plex List

DATA ORG With Addend

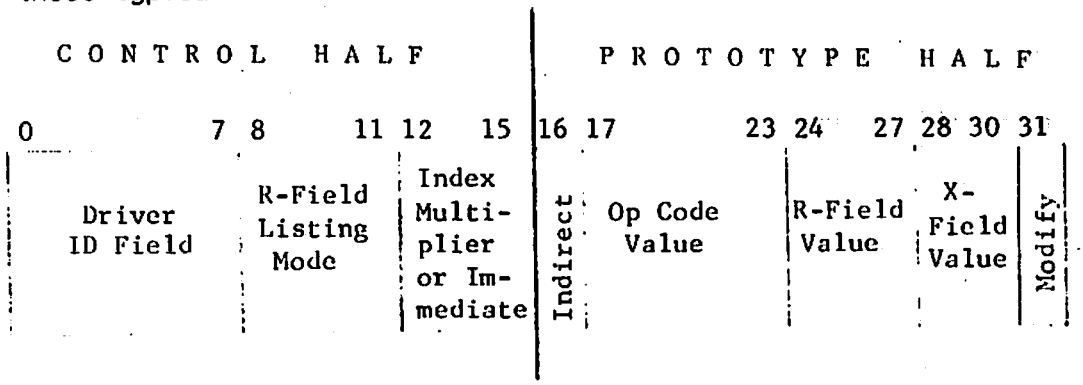
DATA Constant

Return to Program Location (after DATA)

Of these the bread-and-butter indicator is Normal Op Code, which handles the great bulk of machine instructions. Calling Sequence Op Code is very similar, but it lists the Command Field (op code) differently. Not only is the op code listed differently (LCFI becomes SNGL), but the R field is listed with interpretation of the P, C, and E bits rather than as a number.

All of the typical instruction indicatos are followed by a pointer word for the operand. This may be a Symbol Table pointer, a pointer to one of the label lists, a pointer to one of the constant lists, a temp pointer, an external reference name, an absolute address pointer or an address function plex pointer. For details of how these operand pointers are printed on LO, see the routine Print Operand (PROPERND), described in 2.29.3.3.1.

One of these typical instruction indicators looks like this:



The right-hand half (with the exception of bit 31) becomes the left half of the generated instruction. Bits 12-15 exit mainly for the benefit of ADRCALC in Pass 2, but a zero value in this field causes OUTPROG to regard its operand as a 20-bit field instead of a 17-bit field. Thus an index multiplier of zero implies an immediate instruction.

Bit 31 implies that the instruction modifies memory, a consideration that is important only to the FORTRAN Debug Package in Pass 2. The R-Field Listing Mode (bits 8-11) may have the value 0, 1, 2, or 3, meaning

0. List R-Field unsigned (0 to 15)
1. List R-Field signed (-8 to +7)
2. List R-Field with P, C, and E
3. Omit R-Field

2.22.1 Some Typical Instruction Drivers

Four rather typical instruction outputters differ only in the way they print the Op Code. These are Output Normal Op Code (OUTOPCD), Output Calling Sequence Op Code (OUTOPCS), Output Branch Op Code (OUTBOPC), and Output Branch/Z Op Code (OUTBOPCZ). The routines just named handle the whole instruction, not just the op code. Each of them ADRs an Op Code printing routine address, then branches to Output Normal Instruction (OUTNORM).

OUTNORM stores away the indicator word in NDICWORD, JRSs to NXINWORD for the operand pointer, which it puts in OPERAND. Then it unpacks the indicator word into several different fields and stores these in temps, including one called HEXWORD, which will contain the eight digits that will be listed as the contents of the generated word. It then picks up OPERAND and calls Evaluate Pointer (EVALPNTR, in PASS3F) to figure out what it means and to get a numerical value for it. This numerical value is ORed into HEXWORD.

Now we pick up HEXWORD, JRS to BODATUM (2.28.2), then test the LO Trigger. If this is not set, we skip quite a lot of the following code. If it <u>is</u> set, we call PRLOCHEX, which means "Guarantee Object: Print Location 'P', Hex and Label Field." (Guarantee Object means that if we are not in Object mode, we upspace one line on the printer; there is a trigger (OBJTRIG) that keeps track of whether we are in source mode or object mode; this lets us put out a blank line between source lines and object lines.)

Now it is time to print the location counter, followed by the letter 'P' (for Program location). Then we print the hexadecimal value; this is done in three fields, and the word is unpacked into them by UNPK3HML (Unpack 3 Hex Fields ML). Then we fall into Print Label Field If There Is One (PRLABELF), which checks Label Pointer (LABELPTR) for zero. If it is zero, then there is no label for this line. If it is nonzero then it is either a label pointer (to the LOCLBLST, NLCLBLST, or CRELBLST) or a SPROG pointer (to the SYMTABLE). The first three cases get printed as an integer followed by S, S, or G; the last case gets printed as a name.

Now it is time to print the Op Code. We TAB to the Op Code Column (OPCODCOL), then JRS *PO, which takes us to one of four Op Code printing subroutines. Print OPCD Mnemonic (PROPCD) loads OPCODVAL into the index register and uses it to address a 128-item table of 4-character mnemonics called OPCDTB. Print OPCS Mnemonic (PROPCS) uses OPCODVAL as a key to search the OPCSLIST, a static list of eight items. If the thing is not found, the mnemonic "----" is printed. Print BOPC Mnemonic (PRBOPC) uses the combined op code and R field to index a 32-item table of Branch mnemonics. Print BOPCZ Mnemonic (PRBOPCZ) calls PRBOPC), then prints a letter Z and exits.

Now we are finished with whatever routine was called through PO, and we are ready for the R Field: we pick up RFMODE (given to us by UNPAKNML) into X1/and JRS *LISTR,X1 which will lead us to one of four R-Field printing routines, described earlier. Then we check the INDTRIG and if the instruction is indirect, we TAB to the operand column minus one and print an asterisk. Then regardless of the asterisk, we TAB to the operand column and call Print Operand, Addend, X Field (PROPRNDX), then call Print Comment (PRCOMNT). Now we rejoin the code that branched if no LO. We zero the ADDFND (for the next

instruction), add one to LOCCOUNT, then Ditch One and Exit. The thing we ditched, of course, was the address of the Op Code printing subroutine.

2.22.2 Some Rinky-Dink Drivers

The drivers just described are impressive becuase of how much work it takes to handle them. What they do is not impressive, just hard. The drivers in this section are just the opposite--pathetically easy to do, although their results can be impressive.

Take for example Output Comment Next Line (OUTCNL), which handles the output of the CNL POP. This consists of two instructions:

```
OUTCNL   STO      COMNTLOC
         BRL      EXITML
```

The main reason this driver is so simple is that the real work, printing the comment, is done elsewhere. Normal instruction outputters, such as those just described in 2.22.1, examine COMNTLOC to see if it is zero and, if not, print the comment so addressed. OUTCNL does not care if COMNTLOC already has a comment location it it: if Pass 2 has the inclination to try to print two comments on the same line, the first one just will not get printed.

Then there is Output Addend (OUTADND). This consists merely of

```
OUTADND  ADS      ADDEND
         BRL      EXITML
```

Here if multiple addends are applied to the same instruction, they just accumulate. The next actual generative driver will incorporate the addend into the instruction, then zero ADDEND.

Output End of Program (OUTEND) just METs the End Trigger (ENDTRIG) and clears one and exits (discarding the End Indicator).

The final driver I wish to discuss in this secgion might be called borderline-rinky-dink. It is Output Label Definition (OUTDEF). First of all, it is there only for the benefit of the Object Listing (LO): the actual work of definition has been done by Assemble Code (ASMCODE) in Pass 2. So if the LO Trigger is off, we discard the label pointer and forget the whole thing. But assuming there is LO, we have the problem of multiple labels defined at the same point. With multiple addends at the same point (an unlikely occurance anyway), we just add them up; with multiple comments we just ignore all but the last. But there is no such easy out for multiple labels.

We have a temp called LABELPTR that holds the pointer to the label to be defined on the next line. If there is none, naturally this is zero. When we

come into OUTDEF, we check LABELPTR for zero; if it is not zero, then there is already a label waiting to be defined. So we do some remarkable things. While holding the new label pointer in WD, we Guarantee Object Mode (GTOBJECT, see 2.22.1), then print the location counter followed by the letter P (PRPLOC); then we JRS to Print Label Field (PRLABELF, a routine that is just fallen into by most of the outputting routines) which prints the label in LABELPTR. This may be a Symbol Table label, or an S, G, or $ label. Then we TAB to the Op Code Column and print EQU, then to the operand column and print S. Then we output the print line and STO the new label pointer in LABELPTR where it will be ready for printing by the next generative driver. Or it may be printed on an EQU S line by OUTDEF if still another label definition comes along.

2.22.3 Output Immediate Instruction (OUTIMMOP)

This driver is poorly named, for it is not the only routine that can output immediate instructions (Output Normal Op Code, OUTOPCD, can do it too). What this routine really does is to process the output of the BIM POP, which is only one of several possible ways of generating immediates.

The peculiar thing about the output of the BIM POP is that the operand word (the second word) will be a pointer to one of the constant lists, yet the generated operand will be the value of that constant, not the location of that constant. This is at variance with the usual rules of Pass 2/Pass 3 interface.

A more stratightforward approach might be for Pass 2 to take the value of the constant and convert it into an absolute address pointer. This would work fine as far as the binary output is concerned (indeed, it is sometimes done), but it would leave Pass 3 in some doubt how to print the value of the constant on LO. The temptation would be to list it as a decimal integer, but the constant may be intended as logical or hexadecimal, or maybe Hollerith or real. For this reason, we send along the constant pointer and list it just as if it were a literal without the "=" sign.

2.23 I/O Unit Reference

I/O units are referenced in terms of their external DCB names; the names corresponding to FORTRAN I/O unit numbers consist of "F:" followed by the decimal expansion of the unit number in EBCDIC. Thus, units 13 and 108 have DCB names "F:13" and "F:108", respectively. At run-time, each I/O unit to be used must have a DCB built for it; the system loader does this in response to (1) REFs of such names in the object code, (2) ASSIGNs in the job stream. The compiler tries to reduce the number of !ASSIGNs the user will have to include at run-time by REFing the DCB names it knows will be needed, and a few that it assumes might be needed. When the unit number is not a constant at compile-time, or when it is a nonstandard unit (i.e., not 101, 102, 103, 104, 105, 106, or 108), the corresponding DCB name is established at run-time by the library subroutine 7UNITADR called from the various I/O driver routines. DCBs for these units must be ASSIGNNed by the user at run-time. There is one other reason why the compiler may REF certain DCB names: the system loader will not produce DCBs for names REFed in the library. This means that the names of all known DCBs must be REFed in the FORTRAN object code, even if they are only used within the library. Thus, in a main program compilation, we REF "F:108", because 9STOP and 9PAUSE need it, "M:DO" because the error reproting routines need it, etc.

The actual REFing is done in Pass 2 by putting the eight-character names of things to be REFed on the REFLIST. The subroutine Register REF Name (RGREFNAM) is given a "link" (i.e., the address of the first word of a two-word item) to the name. It searches REFLIST for that name and, if it was not already there, puts it on the list. When Pass 3 comes along, it generates REFs for all of

the names on REFLIST (see 2.28.1). REFLIST collects the names of library subroutines called from the object code (ASMCODE supplies these names when it encounters P23BLC interface items), and DCB names supplied as follows:

1. Initially (in Pass 2 initialization: PROGGEN), if the compilation is of a main program (MAINPTRG is set), we call RGREFNAM to register the following DCB names needed by the library:

   F:108 standard listing unit

   M:DO diagnostic output unit

   M:OC operator's console

   If, furthermore, the DEBUG option is given, we generate the following additional DCB names needed by FDP (FORTRAN Debug Package) library routines:

   F:UF User File

   M:SI source input (debug commands)

2. During Pass 2, when BCD I/O statements are encountered (BCD READ and WRITE, and INPUT), the subroutine Check Unit (CHEKUNIT) is called; it will register zero, one, or a group of REF names as follows:

   - If the unit number is one of the standard unit numbers (101,...,106, 108), we REF that one unit's DCB name;
   - If the unit number is a constant other than one of the standard unit numbers, nothing is REFed;
   - If the unit number is not a constant, we REF DCB names of all the standard input units (101, 103, 105) for a READ or INPUT statement, the standard output units (102, 104, 106, 108) for a WRITE statement.

2.24 Comments in Object Listing

As is described in Section 2.29, much thought has gone into making the object listing as informative and readable as possible. Occasionally the compiler generates object code whose purpose is not obvious, if not totally inexplicable. Rather than let the user wonder why this strange code has been used, the mechanism was included to print comments on the object listing. Here are two examples:

```
IF (J**K) 2,4,2
```

```
                LW,9    J
                MW,9    K
                AI,9    0               COMPARE WITH ZERO
                BEZ     4S
2           A = AINT(B)
                LW,8    B
                FAS,8   =X'47000000'    TRUNCATE FRACTIONAL PART
                STW,8   A
```

It is probably not obvious to the casual Sigma user that integer multiply does not set the condition codes in the expected manner, or that AI 0 is the same as CI 0, but faster. And it must be inexplicable why we would want to use an unnormalized floating point zero, if you do not know that this is the "magic" value that truncates the fractional part of floating point values.

It is the responsibility of Pass 2 to decide when it is generating something unclear, and to issue a CNL POP (Comment Next Line) just before generating the instruction in question. The CNL addresses a quote (surrounded by dollar signs). It send the address of the quote along to Pass 3. The quotes themselves are in P2P3ROOT. When Pass 3 receives one of these, it remembers that fact and prints the comment when it has finished printing the op-code and operand. (This is done in PRCOMNT, Print Comment and End the Line, in the assembly OUTPROG.)

Occasionally it is difficult for Pass 2 to generate the comment ahead of the instruction that it applies to. For example, the instruction may be subscripted and cause subscript code to come out first, and we do not want the comment to come out on the subscript code. In such cases (and there are only a few of them), Pass 2 can do the CNL after it generates the instruction, and then call PREVINST (Apply Comments and/or Addends to Previous Instruction) to move the comment into the right place.

2.25 Subprogram Calling and Receiving Sequences

2.25.1 General Considerations

There are two kinds of calling/receiving sequences to be considered: standard and nonstandard. Standard calling sequences are used for calling subroutines and functions that can be written in FORTRAN source language. They must be able to handle various numbers and types of arguments. In general, the arguments that are passed are addresses rather than values. (Results of FUNCTIONs, however, are values, not addresses.)

Nonstandard calling sequences are used to call system routines, which must be coded in machine language and generally accept a fixed number of arguments (typically one) of a prescribed type. Arguments to nonstandard subroutines are usually values rather than addresses.

2.25.1.1 Mathematical Functions: Drivers and Evaluators

The compiler recognizes many function names as intrinsic functions. Some of these are generated in-line. Others are performed with a special call. For example, when the compiler sees a call on the SQRT function it does not generate a standard calling sequence to a routine called SQRT, but instead generates a nonstandard call to a routine called 9SQRT. 9SQRT receives its argument in a register and is called in a one-word calling sequence instead of a three-word sequence. 9SQRT can run fast, since its arguement is where it ought to be and it does not have to decipher a complicated calling sequence. The object program calling 9SQRT is also shorter.

Since 9SQRT actually computes a square root, it is termed an "evaluator". Since it is called with a nonstandard calling sequence, it is considered a nonstandard evaluator.

There is also a routine named SQRT that does no actual computation, but calls 9SQRT to do its work. SQRT has a standard receiving sequence and is provided so that a user can declare the name SQRT to be EXTERNAL, then pass the function name SQRT in an argument list. Such usage is relatively rare (but is allowed in ANS (formerly USASI) FORTRAN IV, as well as IBM FORTRAN IV), but in order to pass the name SQRT as an argument, there must be a subroutine called SQRT with a standard receiving sequence. Therefore, the library contains quite a few routines that are seldom called and do no real work. Since SQRT does no work other than call 9SQRT, the routine is called a "driver"; and since it has a standard receiving sequence, it is called a standard driver. There is a standard driver for nearly every intrinsic function.

Some intrinsic functions (such as ABS) are so simple that the code for them is generated in-line. For such a function, there is no external nonstandard evaluator. Nonetheless, the library must provide a function of that name with a standard receiving sequence. Therefore the ABS routine must do its own work, and is thus called a standard evaluator instead of a standard driver. Most standard evaluators are quite simple. A few, however (such as AMAX which accepts a variable number of arguments), are rather involved. ATAN accepts either one or two arguments and calls either 9ATAN1 or 9ATAN2 (nonstandard evaluators). There is no such thing as a nonstandard driver.

When a function name is intrinsic, it can generally accept arguments of various types. For example, SQRT can accept arguments that are integer, real, or double precision (but not logical), since the compiler can generate the necessary code to convert the arguments.

However, when a function name is declared EXTERNAL and is no longer intrinsic, it becomes much more of a burden to allow arguments of the wrong type because

the function must then, at execution time, inspect the types of its arguments and convert them accordingly. This overhead would have to be paid, even though most of the arguments passed are of the right type. However, Xerox Extended FORTRAN IV does not pay this overhead; standard drivers and evaluators accept only arguments of the "correct" type. This does not mean that it is not possible to write a function or subroutine with a standard receiving sequence that accepts arguments of varying types. Enough information is passed in the calling sequence so that such a routine can indeed be coded in machine language. It simply means that the standard evaluators and drivers in the library are not coded this way.

(Nor are such standard subroutines as SLITET and EOFSET coded this way.)

2.25.1.2 Naming of System Routines

FORTRAN system subroutines must have names that cannot conflict with FORTRAN indentifiers. A FORTRAN identifer begins with a letter; its other characters, if any, are letters and digits. Thus, a system subroutine name must be anything other than such an identifier.

In the FORTRAN IV Library, the prefix digits 9, 8, 7, and 6 are used. Nearly all system-routine names begin with the digit 9 (e.g., 9SQRT, 9SETUPN). Such names beginning with 9 are entry points. Names beginning with the digit 8 (e.g., 8SENLITE) are system temps used to communicate between system routines; these are not called, but are loaded or stored.

The third class of system names is the group beginning with the digit 7. Like the names beginning with 9, these are entry points, but nonstandard entry points. For example, the routine 9EXP (for raising e to the X power) has the nonstandard entry point 7EXP, which is called 9SINH (hyperbolic sine). 7EXP does part of the work done by 9EXP, and 9SINH needs the same piece of partial work done. Generally, the "7" entry points are important only to those library routines that call them. The machine-language user will probably not want to call them (though he can if he wishes), whereas he probably will want to call the 9-routines. The 7-names are not referenced in compiler-generated code; the 9-names are.

Names beginning with the digit 6 are temps, much like those beginning with 8. However, whereas the 8-temps communicate between routines, the 6-temps are local and are associated with only one routine. They are external only for the purpose of re-entrance.

2.25.2 Standard Calling/Receiving Sequences

2.25.2.1 Standard Calling Sequences

Standard calling sequences are most easily defined by an example that shows FORTRAN source code and its associated object code:

```
REAL Y(10,10), R

LOGICAL L

INTEGER K

CALL URN (Y,K,L,R)

     LI,14  4  ARG COUNT TO 14 (NA)

     BAL,15        URN      LINK THRU 15(LC)

     SNGL      Y

     INTG      K

     LOGL      L

     SNGL      R
```

The number of arguments (in this case four) is loaded into register 14; then the subroutine is called, linking through register 15. The words that follow the BAL specify the arguments. The operation code portion of such a word indicates the arithmetic type of the argument (SNGL = single precision real, etc.) and the address is the 17-bit word address of the argument. Note that even for doubleword arguments, the 17-bit word address is passed, not the 16-bit doubleword address. No provision is made for passing addresses other than word addresses, and this scheme is not intended to handle byte or halfword quantities.

A call with no arguments is allowed and is performed in the following manner:

```
CALL LOP

     LI,14          0        ARG COUNT TO 14 (NA)

     BAL,15         LOP      LINK THRU 15(LC)
```

Sometimes an address passed as an argument must be computed at execution time, e.g., A(J), where A is an array. It is considered bad for reentrancy to store such a computed address directly into the calling sequence, so an indirect bit is provided in all calling sequence words. For example:

```
INTEGER J

REAL A(20)

CALL QT(A(J))
```

```
        LW,9    J
        AI,9    A-1
        STW,9           1TEMP

        LI,14           1       ARG COUNT TO 14 (NA)
        BAL,15          QT      LINK THRU 15 (LC)
        SNGL    *1TEMP          ADDRESS INDIRECTLY THRU 1TEMP
```

An additional bit is provided in the operation code portion of a calling sequence word, to indicate that the argument is "protected" (i.e., that it should not be stored into). This bit is set on arguments that are constants or nontrivial expressions. As far as the object listing is concerned, the fact that this bit is set is indicated by placing a 'P' in the register field. For example:

```
    CALL URN        (Y, J+K, .TRUE., -3.78)
        LW,9    J
        AW,9    K
        STW,9           1TEMP

        LI,14           4       ARG COUNT TO 14 (NA)
        BAL,15          URN     LINK THRU 15 (LC)
        SNGL    Y
        INTG,P          1TEMP   NOT INDIRECT BECAUSE 1TEMP CONTAINS VALUE,
                                NOT ADDRESS
        LOGL,P          =.TRUE.
        SNGL,P          =-3.78
```

The protected bit is tested in input routines, to keep from changing these values. A machine-language user who wishes to use protected bits may define

```
    P   EQU 8
```

before using P in the register field.

2.25.2.1.1 Subprogram Names and Statement Labels as Arguments

So far, we have considered arguments that are

1. scalar variables
2. array elements
3. array names
4. constants (other than literal)
5. nontrivial expressions

It has been shown that when an array name is written as an argument, the address of the first element of the array is passed; thus, the generated code is the same as if the first element of the array had been written instead of the array name. We have not yet considered arguments that are themselves dummies, nor have we dealt with

1. subprogram names
2. statement labels

When a subprogram name is passed as an argument, the address of the entry point of the subprogram is passed. For example:

```
EXTERNAL  URN

CALL QT (URN)

     LI,14          1          ARG COUNT TO 14 (NA)

     BAL,15         QT         LINK THRU 15(LC)

     PZE,P     URN
```

Notice that the calling sequence word has the protected bit set, since it is injudicious to store into a subprogram entry point. Notice also that, since URN is a subroutine name rather than a function name, it is passed without type (as indicated by the PZE op-code). If URN had been a function of, say, real type, the op-code would have been SNGL,P instead of PZE,P.

Statement labels are passed in a different way. When 23S is used as an argument, it is not the location of the label 23S that is passed by the location of a branch instruction (B) to that label. Thus, for example:

```
CALL QT (23S)

     LI,14          1          ARG COUNT TO 14 (NA)
```

```
        BAL,15      QT          LINK THRU 15 (LC)
        PZE,P       1G
        .           .
        .           .
        .           .
1G      B           23S         BRANCH LEADING TO 23S
```

This is done so that statement labels passed as arguments will work the same way as variables to which labels have been ASSIGNed. The statement ASSIGN 23 TO K causes a B instruction leading to 23S to be stored in the variable K. Thus, CALL QT (K) has the same effect as CALL QT (23S). This also makes it possible to write

```
    SUBROUTINE QT (LBL)

    INTEGER LBL, J

    J = LBL
```

The effect on the variable J of the statement J = LBL is the same as ASSIGN 23 TO J. This will work properly regardless of whether QT was called with K or 23S as the argument.

One anomaly is caused by this method of passing statement labels; this is in the LOCF function. For all other arguments, LOCF returns, as an integer, the absolute word address of the argument. Thus, for a subprogram name as the argument, LOCF returns the location of the entry point; but, for a statement label, LOCF does not return the location of the label but the location of a branch to that label.

2.25.2.1.2 Arguments That Are Themselves Dummies

When a subprogram argument is also a dummy, the effective address received by the current subprogram must be passed on to the called one. In the case of dummy scalars and subprogram names, this can be handled quite conveniently with the indirect bit in the calling sequence. For example:

```
    SUBROUTINE LOX (SCALAR,SPROG)

    EXTERNAL SPROG

    .
```

```
.
.
CALL YAZ (SCALAR,SPROG)
     LI,14          2              ARG COUNT TO 14 (NA)
     BAL,15         YAZ       LINK THRU 15 (LC)
     SNGL *SCA*SCALAR
     PZE       *SPR*SPROG
```

As explained below in the section on "Standard Receiving Sequences", 2.25.2.2, the temps SCALAR and SPROG contain the addresses of these dummies. In passing dummy array names and dummy array elements as arguments, it is usually necessary to calculate an address:

```
     SUBROUTINE LAT (ARRAY)
     REAL ARRAY (10)
     .
     .
     .
     CALL KAP (ARRAY,ARRAY(J))
          LI,9      1
          AW,9      ARRAY
          STW,9          1TEMP
          LW,9      J
          AW,9      ARRAY
          STW,9          2TEMP
          LI,14          2              ARG COUNT TO 14 (NA)
          BAL,15         KAP            LINK THRU 15 (LC)
          SNGL      *1TEMP
          SNGL      *2TEMP
```

The temp ARRAY contains the address of ARRAY (0).

An element of a multiple dummy is rather awkward to pass as an argument:

```
        SUBROUTINE ZAGREB (/MULDMY (N)/)
        INTEGER MULDMY, N, LADO
        .
        .
        .
        CALL SOLISTI (MULDMY(LADO))
             LW,1     LADO
             LW,1     *MULDMY,1
             BGEZ     1G              BRANCH IF ARG NOT INDIRECT
             LW,1     0,1             HANDLE INDIRECT
    1G       STW,1         1TEMP
             LI,14         1              ARG COUNT TO 14 (NA)
             BAL,15        SOLISTI        LINK THRU 15 (LC)
             INTG     *1TEMP
```

2.25.2.2 Standard Receiving Sequences

Standard receiving sequences are quite similar to standard calling sequences and are best defined by example:

```
        SUBROUTINE URN (W,M,B,Q)
        REAL W(10, 10),Q
        LOGICAL B
        INTEGER M
             URN      LI,5     4          DUMMY COUNT TO 5 (ND)
```

```
                BAL,6     9SETUPN   LINK THRU 6 (LR)
                SNGL      W
                INTG      M
                LOGL      B
                SNGL      Q
                STW,15    9PTMP     SAVE EXIT LOC
                LI,9      -11
                AWM,9     W         OFFSET THE ARRAY W
                .
                .
                .
W               RES       1         TEMP FOR DUMMY W
M               RES       1         TEMP FOR DUMMY M
B               RES       1         TEMP FOR DUMMY B
Q               RES       1         TEMP FOR DUMMY Q
1PTMP           RES       1         TEMP FOR EXIT LOC
```

The system subroutine 9SETUPN moves the argument addresses from the calling sequence into the temps (W,M,B, and Q) provided for them in the receiving program. It also adds the argument count (register 14) to the link address (register 15) so that a return from the subroutine URN will lead to the location just after the calling sequence in the source program. A RETURN statement in the above subroutine would be generated as

```
        B       *1PTMP
```

A RETURN statement in a FUNCTION subprogram would contain a load instruction before the branch, to pick up the result of the function into the appropriate register. Aside from this difference, the entry and exit from SUBROUTINEs and FUNCTIONs is exactly the same.

References to the dummy scalars M, B, and Q are made indirectly through the temps M, B, and Q. For example:

```
    M = M/73
```

```
LW,9 *M

DW,9 =73

STW,9    *M
```

The address passed to URN for the array W is the address of W(1,1). In addressing W, it is more convenient to deal with the address of W(0,0) than W(1,1) so the code following the receiving sequence, namely

```
LI,9 -11

AWM,9     W          OFFSET THE ARRAY W
```

converts the received address of W(1,1) to that of W(0,0). The distance between the first element of an array (in this case, W(1,1) and the zeroth element (the element addressed when all subscripts are zero) is called the offset (in this case -11).

Elements of the array W are referenced through the temp W, indirect and indexed. For example:

```
W(J,K) = 0

    LI,8 0

    LI,1 10

    MW,1 K

    AW,1 J

    STW,8    *W,1
```

The "protected" bit may be used in receiving sequences as well as calling sequences. When the protected bit is set on an argument in a calling sequence, it indicates that the argument should not be stored into. When the protected bit is set on a dummy in a receiving sequence, it indicates that the dummy is stored into; hence the argument should not have its protected bit set. Currently the compiler never generates protected bits in receiving sequences.

In the nondebug mode, the subroutine 9SETUPN is not concerned with the protected bits in a calling/receiving sequence, nor is it concerned with the type bits. 9SETUPN does, however, take care of indirect bits in the calling sequence.

A wrong number of arguments will never cause either 9SETUPN or the subroutine URN to return to the wrong placee. The number of dummies determines where 9SETUPN shall return, and the number of arguments determines where URN shall return. Suppose there are n arguments and m dummies. If there are too many

arguments, then only the first m of them are set up; the others are ignored. If there are too few arguments, then only the first n dummies are set up. The other dummies remain set to whatever they were on the previous call to URN. If there was no previous call on URN, the results of referencing an un-set-up dummy are unpredictable. Such usage is illegal.

The register conventions for 9SETUPN are as follows:

0	Undisturbed
1-4	Destroyed by 9SETUPN
5	Dummy Count
6	Link register for 9SETUPN
7-9	Undisturbed (may receive dummy addresses)
10-13	May be destroyed by 9SETUPN
14	Argument count
15	Link register for URN. 9SETUPN adds the contents of 14 to 15 before returning. This is to skip over the arguments in the calling program.

The fact that index registers 7-9 are undisturbed by 9SETUPN can be useful. Typically, the argument addresses received are stored into temps corresponiding to the dummies. The machine-language programmer may decide not to receive an argument address in a temp but receive it in register 7-9 instead. This makes it quite easy to code standard function drivers. For example:

```
SQRT LI,5 1          DUMMY COUNT TO 5 (ND)
         BAL,6   9SETUPN   LINK THRU 6 (LR)
         SNGL 7            RECEIVE DUMMY ADDRESS IN 7
         LW,8 0,7          VALUE OF DUMMY TO 8 (AR)
         LW,6 15           MOVE LC INTO LL
         B       9SQRT     SIMULATE DIRECT CALL ON 9SQRT
```

The instruction sequence

```
LW,6 15
B       9SQRT
```

above is preferable to the more obvious

```
BAL,6     9SQRT
B         *15
```

for two reasons:

1. It is faster.

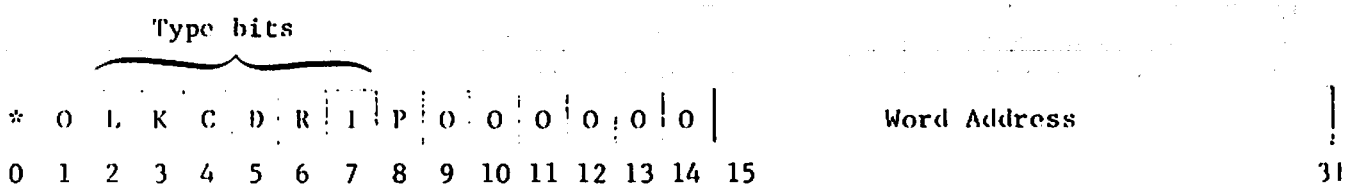
2. It causes error messages printed by 9SQRT to cite the location from which SQRT was called in the user's program, rather than the location from which 9SQRT was called in SQRT.

2.25.2.3 Register Conventions in Standard Subprograms

Once a standard subprogram has gone through the protocol of setting up its dummies, it can destroy the contents of any registers it wishes other than register 0. It should keep the contents of 15 for exiting purposes. Function results are returned in standard registers (see below).

2.25.2.4 Word Format in Calling and Receiving Sequences

The words used to specify arguments and dummies are arranged in the following way:



The operation code portion of the word contains six type bits (bits 2-7), which are allocated as follows:

7 Integer

6 Real

5 Double Precision

4 Complex

3 Double complex

2 Logical

A calling sequence typically has one of these bits set, indicating the type of the argument. A receiving sequence word typically has one or more of these bits set, indicating the types of arguments it will accept. When a statement label is passed as an argument, none of the type bits are set; this means that it can be received by a dummy of any type. The same is also true of a SUBROUTINE name (but not a FUNCTION name) passed as an argument. Usually, a receiving sequence word has exactly one type bit set, but there is one case in which all the bits are set: a dummy SUBROUTINE name. A dummy SUBROUTINE can accept as its argument the name of a SUBROUTINE or the name of a FUNCTION of any type. Therefore, the receiving sequence word for a dummy SUBROUTINE has all six bits set. There will also be one case in which a receiving sequence word will have no bits set: an asterisk in a dummy list, which is intended to receive a statement label. This dummy will conflict with any argument except a statement label or a SUBROUTINE name.

Bit 8 is called the "protected" bit. In a calling sequence word it indicates that the argument should not be stored into; in a receiving sequence word it indicates that this dummy is stored into; hence the argument should not be protected. The protected bit is part of the R field and is listed as such on the object code listing. For calling/ receiving sequences, the compiler will observe the following mnemonics on the object listing.

Not Protected		Protected		FORTRAN Type
PZE	000	PZE,P	008	No type
INTG	010	INTG,P	018	Integer
SNGL	020	SNGL,P	028	Real
DOUB	040	DOUB,P	048	Double precision
CMPX	080	CMPX,P	088	Complex
KMPX	100	KMPX,P	108	Double complex
LOGL	200	LOGL,P	208	Logical
EVRY	3F0	EVRY,P	3F8	Every type

The mnemonic KMPX is derived from the fact that throughout the system the one-character abbreviaition for double complex is "K" (as in Komplex). This letter is used in some library routine names and in the compilation summary.

Bits 15-31 are the 17-bit word address of the argument (calling) or the dummy temp (receiving). Arguments may be indirect, indicated by setting bit 0; dummies may nbt be indirect. Argument addresses are presumed to be memory word addresses and should not be registers. A dummy temp address is typically a memory address but may also be register 7, or 8, or 9. Bits 9-14 are not currently used for anything. They should be set to zero in both calling and receiving sequences. Bit 1 is used to indicate a non-FORTRAN argument (e.g.,

for interface with COBOL) and should otherwise be zero. (Certain COBOL argument types are FORTRAN-compatible; e.g., COMPUTATIONAL-2 type corresponds to REAL type (i.e., SNGL). For such arguments, COBOL does not set bit 1.)

2.25.2.5 Standard Accumulators for Arithmetic

For machines with 16 accumulators like the Sigma 6, 7, and 9, it may seem odd to define "standard" accumulators in which certain types of arithmetic are done. Yet there is good reason to do so. The question arises: in what register do the results come back from functions? Since functions are, in general, compiled separately from the routines that call them, it is necesary to have a convention on this subject. It is generally best that function results come back in a register where they can be used in further computation. Therefore, standard registers for arithmetic of each type are defined as follows:

Type	Register	Register Mnemonic
Integer	9	AI
Real	8	AR
Double precision	8-9	AD
Complex	8-9	AC
Double complex	8-11	AK
Logical	9	AL

2.25.2.6 Returning the Results of Functions

The result of a function of a particular type is returned in the standard accumulator for that type of arithmetic. In generated code, the result is loaded into the appropriate register just before returning. For SUBROUTINEs (as opposed to FUNCTIONs) no result is returned, and the contents of the standard accumulators are not guaranteed after exit from a SUBROUTINE.

Although for most functions the condition codes will be correctly set to indicate how the result compares with zero, the compiler will not assume that the condition codes are correctly set at return time. The compiler will generate a "compare with zero" if necessary to test the result of a function.

This applies not only to functions with standard calling sequences but to intrinsic function evaluators as well.

2.25.2.7 The Argument Set-Up Routines 9SETUPN, 9SETUP0, 9SETUP1, 9SETUP2

The routine 9SETUPN is capable of setting up any number of arguments, including zero. The cases of zero, one, and two arguments, however, are quite common and there are three additional subroutines, 9SETUP0, 9SETUP1, and 9SETUP2 which specifically set up zero, one, or two arguments. They do so significantly faster than 9SETUPN would. The calling sequence to these routines is similar to that for 9SETUPN, but since it would be redundant to pass the number of dummies to these routines, it is not passed. For example:

```
SUBROUTINE WELD (N49)

INTEGER N49

     BAL,6     9SETUP1        LINK THROUGH 6 (LR)

     INTG N49

     STW,15    1PTMP          SAVE EXIT LOC
```

2.25.2.8 Receiving Sequences Involving Multiple Dummies

A multiple dummy is a construct capable of receiving a variable number of arguments. A multiple dummy is neither a scalar nor an array, but it has some of the characteristics of an array. A dummy list in a SUBROUTINE or FUNCTION statement may contain, at most, one multiple dummy. There may be other dummies in the list and, if so, the multiple dummy must come last:

```
SUBROUTINE UBANGI (DAVE, MARIDA,/MULDMY (NN)/)

DOUBLE PRECISION DAVE

INTEGER MARIDA,NN

REAL MULDMY

     LI,5      -3        -DUMMY COUNT (INCLUDING MULTIPLE DUMMY) TO 5 (ND)

     BAL,6          9SETUPV   LINK THRU 6 (LR)

     DOUB      DAVE

     INTG      MARIDA

     SNGL      MULDMY

     STW,15         1PTMP     SAVE EXIT LOC
```

```
        STW,4          NN        STORE MULTIPLE DUMMY ARG COUNT FROM 4 (NMD)
```

The scalar NN is not itself a dummy, although it appears in the dummy list. It simply receives the number of arguments that corresponds to MULDMY. If UBANGI were called with five arguments, NN would be set to 3. The arguments corresponding to MULDMY are not moved into the receiving subprogram. Instead, into the temp "MULDMY" is stored the address of the zeroth argument corresponding to MULDMY. Thus, if UBANGI is called with five arguments, then MULDMY points to the word in the calling sequence for the second argument (although the third argument is the first one that corresponds to MULDMY).

The potential indirectness of DAVE and MARIDA is handled by 9SETUPV, but the potential indirectness of the other arguments is handled at each reference to MULDMY.

```
    MULDMY(J) = 5

        LW,8      =5.0

        LW,1      J

        LW,1      *MULDMY,1

        BGEZ      1G              BRANCH IF ARG NOT INDIRECT

        LW,1      0,1             HANDLE INDIRECT

    1G  STW,8         0,1
```

This kind of receiving sequence is used in generated code and also in library functions such as MAX.

2.25.2.9 The Argument Set-Up Routine 9SETUPV

The routine 9SETUPV is used in receiving sequences that involve multiple dummies, i.e., subprograms that can receive a variable number of arguments. Registers 14 (NA) and 15 (LC) are the same as for 9SETUPN. In register 5 (ND) is passed minus the number of dummies (including the multiple dummy). In register 4 (NMD) is returned the number of arguments that correspond to the multiple dummy (this number may be zero). The fixed dummies (i.e., those other than the multiple dummy) are handled the same way as those in 9SETUPN. Into the temp for the multiple dummy is stored the address of the zeroth calling sequence word corresponding to the multiple dummy.

The type bits and protected bit in the receiving sequence for the multiple dummy apply to all the arguments that correspond to the multiple dummy.

2.25.2.10 The Argument Set-Up Routine 9SETUPM

9SETUPV is sufficient to handle the requirements of code generated by the compiler, but is sometimes not sufficiently flexible for machine-language library routines. Some library routines need to do three things that are impossible with 9SETUPV:

- Allow the variable arguments to be of different types.
- Allow some of them to be protected, but others not.
- Set a maximum on the number of variable arguments.

Consider, for example, the library subroutine EOFSET, which can be called in three ways:

1. With no arguments.
2. With one argument, which is a variable of any type or a statement label (protected).
3. With two arguments, in which case the second must be an integer variable (nonprotected).

For the benefit of such library routines, there is a third kind of 9SETUP routine: namely 9SETUPM. This is very much like 9SETUPN except that it is passed two dummy counts instead of one; these are a maximum and a minimum acceptable number of dummies. The receiving sequence contains a word for each dummy up to the maximum number, and each dummy may have different characteristics. There is nothing akin to a multiple dummy in a 9SETUPM receiving sequence. 9SETUPM returns the number of calling arguments in register 14 (achieved by simply not altering the contents of register 14; the calling program has put it there).

In register 5 (where 9SETUPN receives the number of dummies) 9SETUPM is passed the maximum number of arguments (MXA), and in register 4 it is passed the minimum (MNA). Because of the standard error recovery (in the nondebug mode) from the wrong number of arguments, 9SETUPM can ignore register 4. It simply sets up as many as it can (never more than register 5). Therefore, the nondebug 9SETUPM is identical to 9SETUPN.

2.25.3 Nonstandard Calling/Receiving Sequences

2.25.3.1 Mathematical Function Evaluators

Extended FORTRAN IV uses a set of nonstandard function evaluators whose names begin with the prefix "9". The register conventions are as follows:

Register 0	Unaffected. Library routines never modify register 0.

Registers 1,3,5,7	Preserved by all evaluators.
Registers 2,4	May be destroyed by an evaluator.
Register 6	Library link register. Evaluators are called with a BAL,6 instruction.
Registers 8-15	Used for passing arguments and results. Certain of these registers are also available for use by the evaluators, as detailed below.

Nearly all function evaluators have one argument and one result (the only exceptions are 9ATAN2 and 9DATAN2). Nearly all function evaluators have a result of the same type as the argument (the only exceptions: 9CABS and 9CDABS). Therefore, it is possible to document most evaluators in five groups, based on the type of the argument result. (At the present time, there are no evaluators of integer or logical type.)

Real	8	Argument and result
	9-11	May be destroyed
	12-15	Must be preserved
Double precision	8-9	Argument and result
	10-13	May be destroyed
	14-15	Must be preserved
Complex	8-9	Argument and result (real part in 8, imaginary part in 9)
	10-13	May be destroyed
	14-15	Must be preserved
Double complex	8-11	Argument and result (real part in 8-9, imaginary part in 10-11)
	12-13	May be destroyed
	14-15	Must be preserved
Integer or Logical	8	May be destroyed
	9	Argument and result
	10-11	May be destroyed
	12-15	Must be preserved

The exceptional routines are as follows:

9DATAN2	8	Result and argument P in ATAN(P,Q)
	9	Argument Q in ATAN(P,Q); may be destroyed
	10-11	May be destroyed
	12-15	Must be preserved
9ATAN2	8-9	Result and argument P in DATAN(P,Q)
	10-11	Argument Q in DATAN(P,Q); may be destroyed
	12-13	May be destroyed
	14-15	Must be preserved
9CABS	8	Result and real part of argument
	9	Imaginary part of argument; may be destroyed
	10-13	May be destroyed
	14-15	Must be preserved
9CDABS	8-9	Result and real part of argument
	10-11	Imaginary part of argument; may be destroyed
	12-13	May be destroyed
	14-15	Must be preserved

2.25.3.2 Power Routines

Nine different routines are needed to implement the ** OPERATOR. Routines are provided for raising any number to a power of its own type and for raising any number to an integer power. Register usage in the power routines is as follows:

The power routines and their use of registers 8-15 are as follows:

2.25.3.3 Type Conversion Routines also called Mode Conversion

Although twenty different arithmetic type conversions (such as real to integer) are defined, only six conversion routines are needed. The other conversions are performed by in-line code or by a combination of a conversion routine and in-line code.

The register conventions for registers 0-7 are the same as those for the pwoer routines (described above). The routines, the conversions performed, and the conventions for registers 8-15 are as shown below.

In generated code, the conversions will be performed as follows:

2.25.3.4 Complex Arithmetic Routines

Addition, subtraction, and negation are performed in-line, while multiplication and division are handled by the following routines:

2.26 Double-Word Shenanigans

One hardware feature of the Sigma 5/7 computers that requires consideration from compiler writers is the fact that doubleword instructions (such as Load Doubleword and the Long floating arithmetic) cannot address locations that are odd (whose low-order bit is a 1).

2.26.1 Storage allocation

The problem for FORTRAN people is that the FORTRAN IV language makes no provision for machines with this kind of addressing problem. For example, it allows the EQUIVALENCE of REAL and DOUBLE PRECISION variables with no restrictions as to whether such EQUIVALENCE would cause the DOUBLE PRECISION variable to lie in an odd location. So we as designers were forced to make a decision: should we allow such free-wheeling EQUIVALENCE (the COMMON statement can also produce this difficulty) or should we restrict the language to conform to the hardware? If we were to allow doubleword variables to lie where they liked, we would have to pick them up in two halves, one word at a time; then we would do the long floating point operations on them Register-To-Register. This not only makes for longer object code, but it is slower too: Register-To-Register operations are actually slower in execution than Memory-To-Register operations!

The decision was to restrict the source language rahter than to increase the sizes of the object code and the object code generator.

The most serious effect is in the assembly called ALLOCATE, (see 3.8) which processes the COMMON and EQUIVALENCE statements and the interactions between them. This was modified considerably from the 9300 design. The Sigma 5/7 version includes the concept of "parity preference". A particular variable can "prefer" an even location, or an odd location, or it may have no parity preference at all. Quite an elaborate set of diagnostics can come out of violations of the parity rules. An indication of how tricky it can get is given in the following source language example.

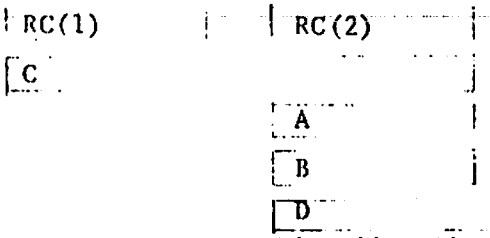
```
DOUBLE PRECISION C, D
REAL A, B, RC(2)
EQUIVALENCE (RC(1),C)
EQUIVALENCE (RC(2),A)
EQUIVALENCE (B,D)
EQUIVALENCE (A,B)
```

These six statements interact in the compiler to produce the following diagnostic error message:

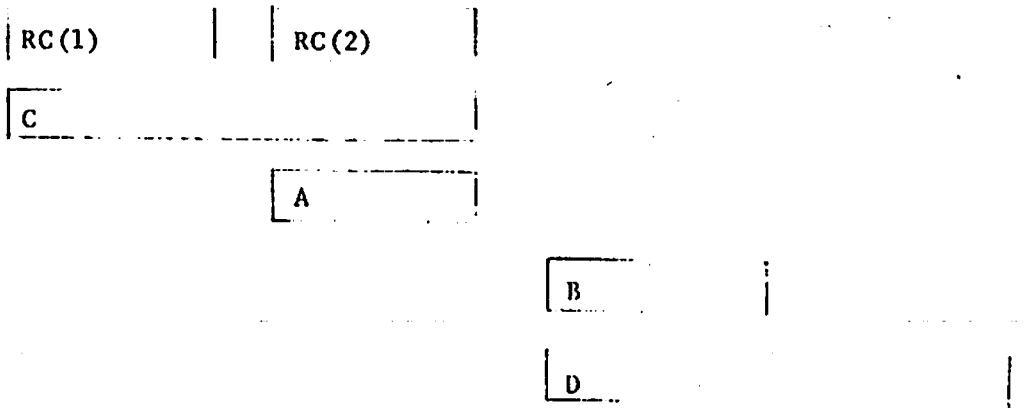
```
WARNING: EQUIVALENCE OF 'A' TO 'B' ON LINE 6 IGNORED; 'C' AND 'D' MUST
         BOTH LIE ON DOUBLEWORD BOUNDARIES.
```

It will take some study on the part of the reader to understand how all six of the above statements are required to produce the error condition. It takes some cleverness in ALLOCATE to keep track of who did what and to whom, but it was the feeling of the compiler's designers that, if we had to introduce this arbitrary restriction, we ought at least to give good messages when the restriction was violated.

The above source example calls for allocation as represented in the following diagram. Here, variables lying below each other occupy the same storage cell.



The compiler refuses to perform this allocation (since the locations of C and D would differ by 1), and instead shifts D upward, as shown in the following diagram. It also shifts B upward, since B is EQUIVALENCEd to D.



2.26.2 Temps

COMMON and EQUIVALENCE statements are not the only place the compiler needed to be careful about doubleword boundaries. It had to be taken into consideration also in the allocation of temps. In the 9300 compiler, when we needed a temp we just allocated one. If the temp happened to be two words long, we bumped the temp count by two. In the Sigma 5/7 compiler, things are not so simple. Now when we want to allocate a doubleword temp, we have to check the temp location counter and make sure it is not odd. If it is odd, we increment it by one. This causes us to waste a word, and in certain pathological statements there could be quite a few words wasted. So we make an attempt to recover the wasted words (see 2.19.4). Pass 2 keeps a cell called SPARTEMP (Spare Temp). When we have to skip over a single word temp in an odd location in order to create a doubleword temp that begins in an even location, we save the location of the wasted word in SPARTEMP. Then, the next time we need a single-word temp, we check SPARTMEP to see if anything is waiting there. (A zero value in SPARTEMP indicates there is no spare temp at present.)

Because of this device, we manage never to waste more than one temp per statement, although the order of reference to the temps comes out looking a bit peculiar.

2.26.3 Constants

The interested reader is also directed to look at the listing of GENLIT (Generate Literals) in Pass 3. Here the compiler needs to be more restrictive and less flexible than the 9300 compiler was in the way it will allow constants to overlap each other. See 2.27.

2.27 Gnerate Literals GENLIT

This is one of the first things done in Pass 3. Up until now, all the constants have been kept on the various constant lists (e.g. ICONSLST, RDCONLST, SCONSLST). This enables the compiler to deal with them more easily in terms of their values. In Pass 3, however, we are not so much interested in their values as in their locations. As yet, they have no locations in the object program. That is what this program does. It goes through the various constant lists and produces the actual literals that will follow the object code. These it puts on the LITERLST. Since the object code has all been generated, we know the size of it, so the location of any constant is its position on the LITERLST plus the generated code size. Pass 3, when listing these constants, will receive from Pass 2 a pointer to the appropriate constant list. From this, it needs to be able to find out the allocated location of the constant. Therefore, GENLIT stores into a word in each entry on each of the constant lists the location at which it has been allocated.

Generating the literals does not just consist of piling all the constants on the LITERLST, for several reasons:

1. Not all constants that appeared in the source program are actually used by the generated code. For example, none of the constants appearing in the following statements ever become literals:

```
K = K + 7

K = K + 8

K = N * 7

K = N * 8

X = Y ** 2

IF (X .LT. 0.0) GO TO 4
```

   They are taken care of (respectively) by MTW, ADM, LI, SAS, FMS, and RLZ instructions. Constants appearing in DATA statements do not need to be generated as literals either. They are generated separately in Pass 2.

   With each constant on the constant lists is an indicator (set up by Pass 2) of whether that constant has ever been used. If it has not been used, it does not need to be generated as a literal.

2. Real and Double Precision constants are both kept on the RDCONLST, and only in Double Precision form. There are two indicators which tell whether each constant has been used as a Real or as a Double Precision entity, or both or neither. Since the rounded Real version of a Double Precision number may not be the same as the first word of the Double Precision version, the constants may actually have to appear separetely on the LITERLST. This is almost always true for Complex and Double Complex constants.

3. We use an overlapped search to try to double up on data words used in constants. For example, the Logical constant .TRUE. and the Integer constant -1 can both use the same literal. We start by putting in the longest constants (Double Complex) and work our way down to the smallest. Alphanumeric String constants go in after K,C,D, and R, but before I, H, and X. We cannot put them in first, because we have to guarantee the double-word constants that they start at an even location. However, we do want to put them in before most of the one-word constants, so that the latter can possibly overlap with pieces of the String constants (such as the preceding character count -- note the peculiarity about String constants that, although they begin with a character count word, their official address is that of the second word, where the string actually begins). Note also that we do not do an overlapped search while putting the String constants

onto the LITERLST, because it is very unlikely that a String constant will overlap a group of floating point constants, and the overlapping of String constants with themselves has been taken care of when they were registered on the SCONSLST (by RGSCONML).

For each type of constant, the procedure is essentially as follows:

1. See if there are any constants on the appropriate constant list. If not, go to the next type of constant.

2. Step through the constant list. For each constant, test if it has been used. If not, go to the next constant. Note that for R/D and C/K, we go through the list twice, once testing for one type and once testing for the other type.

3. Assuming that the constant has been used, get its value into CENTRAL (using as many words of CENTRAL as needed). This may not be the value that appears in the constant list. Real constants have to be converted from Double Precision; likewise Complex from Double Complex.

4. Using the SERK (search with key) POP and a special code word containing key size and item size, search the LITERLST for the value in CENTRAL. If found, use that pointer. If not, create a pointer and put the value onto the LITERLST.

5. The address portion of that pointer, added to the generated program size, gives the location of the constant. Using the original pointer to the constant list, store the new constant location back into the constant's entry on the particular constant list. The place in that entry where the location is to be stored is the same word that formerly held the indicator telling whether the constant had been used. This indicator is no longer needed, because if the constant was never used, no references to it will be coming along from Pass 2. Note that for constants on the RDCONLST (and CKCONLST) there are two indicators (one for Real and the other for Double) and thus two locations to be stored. Often they are the same.

Logical constants are not specially hadnled here, because they are kept on the Integer Constant List. The way the compiler knows whether a constant is Integer or Logical is by the mode field of the pointer to the ICONSLST. Hexadecimal and Hollerith constants cannot be handled that way because they are both integer type, yet they must be distinguishable so that the object listing can show them correctly.

If there are any K, C, or D constants, they must fall on doubleword boundaries, so we set the Literals Even Trigger (LITSEVEN) and increment the object program size (PSIZE) by one if it is odd when we start.

We noted that no overlapped search need be done for String constants. However, there is one other bit of complication. The JPN (Jump and

Process Next Item) POP does not work on the SCONSLST, because the entries are not of constant size. The size of each entry depends on the length of the alphanumeric constant it containss. So we have to do the stepping from constant to constant ourselves. If we are going to put it on the LITERLST then there is no problem; we have to step through the string anyway. But if the constant has never been used (this can happen only by appearing in a DATA statement), we must not just ignore it and jump to the next constant; we have to find out how big it is. The easiest way to do this is to use the code that steps through for the purposes of putting the constant on the LITERLST, but set a trigger that says to just step through, not put anything on the LITERLST.

2.28 Binary Output OUTBO Pass 3

The binary object language produced by the compiler is a subset of that defined in Appendix A of the BPM Reference Manual (Xerox 90 09 54E). The following table shows which Object Code Control Bytes we generate.

Code	Gen'd by FORTRAN	Type of Item
00	no	Padding
01	yes	Add constant
02	yes	Expression end
03	yes	Declare DEF name
04	yes	Origin
05	yes	Declare REF name
06	no	Declare SREF name
07	no	Define field
08	no	Define forward reference
09	yes	Declare DSECT
0A	yes	Define DEF value
0B	no	Declare special CSECT (CSECT 0)
0C	yes	Declare ordinary CSECT (CSECT 1,2...)
0D	yes	Define start address

0E	yes	Module end
0F	no	Repeat load
10	no	Define forward reference and hold
11	no	Supply type information for external symbol
12	no	Supply type and name of internal symbol
13	no	Supply name and forward reference number for undefined symbol
20-23	yes	Add value of declaration
24-27	no	Add value of forward reference
28-2B	no	Subtract value of declaration
2C-2F	no	Subtract value of forward reference
30-33	no	Change expression resolution
34-37	no	Add absolute section
38-3B	no	Subtract absolute section
40-4F	yes	Load absolute (we only gen 44)
50-5F	yes	Load relocatable (long form)
80-FF	yes	Load relocatable (short form)

Object code "expressions" are generated in conjunction with 0A (Declare DEF value), 04 (Origin) and 0D (Define start address) items. All these expressions are representable as "constant plus CSECT or DSECT base"; thus, their object code consists of the following:

(a) If the constant is non-zero, 01 (Add constant) followed by the four bytes of the constant;

(b) 20 (Add value of declaration at byte resolution) followed by the one or two byte number of a CSECT or DSECT declaration;

(c) 02 (Expression end).

Such expressions are always generated at byte resolution.

The assembly OUTBO does the actual generation of the binary object language code. It interfaces with Pass 3 driver routines via four entry points: initially it is entered at STARTBO to gen declarations of all REFs, DEFs, CSECTs, and DSECTs. Then, for each instruction or data word to be generated, BODATUM is called -- it puts out the appropriate load item, and possibly a REF to a previously unREFed library routine name. BOORG (BO origin) may alos be called to gen an Origin item (this is used in handling DATA statements). Finally, FINISHBO is called to gen the Start address and Module end. Whenever OUTBO has accumulated a record full of Object Code Control Bytes, it calls BOWRITE and/or GOWRITE in MERRIMAC to write out the record.

2.28.1 Start Binary Output STARTBO Pass 3

If neither the BO nor GO option was specified, this routine exits immediately. Otherwise, it sets up to generate the first record of object language code, and then performs the following functions:

1. It establishes the four character module identification sequence to be appended to the end of each BO record (if BO is being punched on cards, this identifier will appear in columns 73-76 and a decimal sequence number in 77-80). If the SB name option was specified, the given name will be used (the word containing the four characters is placed in QNAME). If SB was not specified, QNAME is either set to "MAIN", or it is set to the first four characters of the first name in the symbol table (i.e., the name of the first subprogram). Finally, we call INQBUFML (Initialize QBUF, machine language) to convert this name (QNAME) into four 12-bit "column binary" codes which will appear in columns 73-76 of BO records. INQBUFML also initializes the "column binary" codes for the first records decimal sequence number 0000. These eight columns of information are placed in QBUF, a three-word buffer immediately following the main object language buffer BOBUF. When writing a BO record, we send 120 bytes: 108 object code bytes from BOBUF followed by the 12 special bytes from QBUF.

2. Now we output a DEF declaration for the first name in the symbol table, if this compilation is of a subprogram. This DEF is being generated in a manner different from all other DEFs (see item 5(a) below). Ordinarily the two parts of a DEF (namely, the DEF name declaration and the DEF value specification) are output together. Now, if the subprogram being compiled is to be used as a library module, a difficulty arises: in searching a library module for a given name, the loader scans initial DEF and REF name declarations and stops when it encounters anything else (CSECT declarations in particular). Thus this subprogram name must be declared prior to CSECT declarations, but its value may only be specified after the program CSECT has been declared since its value expression must contain the number of that CSECT declaration. See 5(a) below for discussion of how the DEF name declaration is genned.

3. Next we gen declarations of CSECTs for the program and variable sections, known as the P and V sections, respectively. P and V section declarations

are produced by the subroutine GENCSECT which computes the section size (in multiples of eight bytes) and protection type (from PTYPETBL), generates a 0C (Declare CSECT) item, and calls DECLARE to make an item for the CSECT on the Declaration List (DECLLIST). This list consists of a sequence of two-word names in one-to-one correspondence with the object language declaration numbers; whenever we gen a declaration item, we must put the corresponding name on DECLLIST In the case of CSECT generation, the "name" is simply a dummy item consisting of two words of zeros. DECLARE returns the corresponding declaration number, which GENCSECT stores in the appropriate place in the three-word Declaration Number Table (DECNOTBL), used to hold the declaration numbers of the P, V, and BC (blank COMMON) sections. The information in DECLLIST and DECNOTBL is used principally by FINDDECL (see 2.28.2) to determine the declaration number of a given, previously declared symbol or section.

4. The next item to be generated are DSECT declarations for balnk COMMON and all labeled COMMON blocks; the names and sizes of labeled COMMON blocks are obtained from items on the Block Name List (BLNAMLST). Each COMMON block section is declared by GENDSECT, given its size and name (for blank COMMON, we use the name "F4:COM"). GENDSECT works much like GENCSECT, described above; except that it first calls GENDEF (described below) to output a 03 (Declare DEF name) item to declare the name of the DSECT and register that declaration on DECLLIST. Then GENDSECT generates the section declaration (09 item), referencing the name declaration number returned by GENDEF. Now GENDSECT calls DECLARE to register the section declaration, and for blank common, places that declaration number in the third word of DECNOTBL. Note that the generation of a DSECT produces two items on DECLLIST and thus used two declaration numbers, as required by the object language. The two DECLLIST items are identical; it is the first one (corresponding to the name declaration) that will be found by the SER in FINDDECL when looking for a labeled COMMON declaration.

5. Now we make a sweep through the Symbol Table looking for non-local subprogram names (to be DEFed), symbols with EXTREF trait (to be REFed), and symbols with the GLOBAL trait (to be declared as DSECTs). If the compilation is of a subprogram, we handle the first symbol specially, to compensate for the special treatment it got via item 2 above: its name has already been declared, so we proceed directly to the value specification as described under (a) below. All other symbols are tested to see if they fall into one of the three interesting categories:

   (a) If it has the SPROGDEF (Subprogram Definition) and EXTERNAL (non-local) traits, we DEF it. This is done in two parts: first we call GENDEF to gen a DEF name declaration (03 item) and register it on DECLLIST); second, we gen a DEF value item (0A) which references the name declaration number returned by GENDEF. In order to generate the DEF value expression, we first call EVALPNTR (Evaluate Pointer, in PASS3F) to express the DEF value in terms of Offset, Relocation code, and Address Resolution code, as described in 2.28.2. Finally, we call GENEXP (see 2.28.4) to generate the corresponding object code expression.

(b) If the symbol has the EXTREF (External Reference) trait, we REF it by calling GENREF which outputs the Declare REF name (05) item and registers it on DECLLIST.

(c) If the symbol has the GLOBAL trait, we call GENDSECT to gen a DSECT declaration for it, just as for labeled COMMON blocks as described above (4).

6. Having finished the symbol table sweep, STARTBO now puts out REFs for all the names on the REFLIST; these are DCB names (like F:108 and M:0C) put there by Pass 2, and special non-FORTRAN names appearing in in-line assembly code (such as 39GRUNCH or FUNC:APL).

7. STARTBO now generates DEFs for special symbols attached to the P (Program) and V (Variable) control sections. These symbols are generated to assist the programmer in debugging his FORTRAN program. The names will appear on his load map, enabling him to locate the various CSECTs. The main program's P and V sections are identified by the generated symbols "4MAIN" and "V#4MAIN", respectively. For a subprogram, the P section already has a symbol at location zero (the subroutine name itself); for its V section, we create the symbol "V#..." where "..." represents the subroutine name (up to eight characters). Note that this created symbol may be as long as ten characters; this is the only instance in which we need to gen a declaration of a name longer than eight characters. STARTBO gens the DEF of "4MAIN", if appropriate, by calling GENDEF and GENEXPPO (Gen Expression to zero, relative to P section). In DEFing the "V#" name, code similar to that in GENDEF (it precedes the symbol name with "V#") is executed in-line; a two-word zero is used in registering the declaration on DECLLIST in order to prevent the appearance of the sub-program name or "4MAIN" from appearing twice. The "V#" DEF value expression is genned by GENEXP, given the value zero and Relocation code V (relative to V-section).

8. The last item generated by STARTBO is the Origin (04) item. This is done in GENORG, which calls GENEXP to gen an expression for "zero, relative to the P section".

2.28.2 Output BO Datum BODATUM Pass 3

If neither BO nor GO was specified among the options, this routine exits immediately. Otherwise, it generates the load item (44, 50-5F, or 80-FF) necessary to load the given instruction or data word. This value is passed to BODATUM in terms of three quantities: (1) value, passed in W0; (2) Relocation code, passed in index register X1; and (3) Address Resolution code, passed in ADRRES In addition, for some values of the Relocation code, there needs to be a labeled COMMON block name, a GLOBAL name, or the name of an externally referenced symbol; this is passed in IDCENT. This representation is interpreted to mean "the quantity obtained by adding to value the bias associated with the Relocation code, aligned according to the Address

Resolution code". The Relocation code (along with the name in IDCENT, when required) specifies the application of biases as follows:

Code Value	Name Required	Interpretation
0	no	Absolute (no relocation)
1	no	Relative to P-section bias
2	no	Relative to V-section bias
3	no	Relative to blank COMMON section bias
4	yes	Relative to named labeled COMMON bias
5	yes	Relative to named external REF value
6	yes	Relative to named GLOBAL symbol value

The Address Resolution code is interpreted as follows:

Code	Type of address alignment
0	Byte
1	Halfword
2	Word
3	Doubleword

Address Resolution codes other than 2 (Word) can arise only from in-line symbolic code where the programmer has used the addressing functions BA, HA or DA. This three (or four) element representation of an instruction or data word is established by EVALPNTR (Evaluate Address Pointer, in PASS3F).

BODATUM generates the load item as follows:

1. If the Reloc code is zero, a "Load Absolute 4 bytes" item (44) is genned.

2. If the Reloc code is not zero, we call FINDDECL (Find Declaration Number, see 2.28.4). If its declaration is not found, we call GENREF to REF it and give it a number (this happens only for library routine names not previously encountered). In either case, we now have a declaration number. If it is less than hex 40, and if the Address Resolution is 2 (Word), then we gen the "short form" load item with control byte = hex 80 + declaration number. Otherwise, we gen a "long form" load: control byte = hex 58 + resolution code when the declaration number is less than hex 100; control byte = hex 50 + resolution code for larger declaration numbers. The control byte, in this case, is followed by the one or two byte declaration number.

2.28.3 Finish Binary Output FINISHBO Pass 3

If neither the BO nor GO option was given, this routine does nothing. It is called after all the instructions and data items have been generated. For a main program compilation it generates a Define Start Address (0D) item with an expression to zero, relative to P-section (genned by GENEXPPO). It then puts out a Module end (0E) item with the error severity code corresponding to the worst error level encountered during the compilation, as given in the cell ERRLEVEL. Object code error severity values correspond to the FORTRAN errors levels as follows:

FORTRAN err level	Object code Severity	Meaning
0	0	No errors
1	4	Minor errors
2	7	Major errors, no statements deleted
3	10	Major errors, statement(s) deleted

This correspondence is established in the fixed Error Severity Table (ERRSEVTB). Having generated the final object language control bytes, FINISHBO finally changes the BO header identification byte from hex 3C to hex 1C to indicate "last record" and calls BORECML to output the last BO record.

2.28.4 Object Language Expressions/Declarations

Each time a Declaration item is generated (03, 05, 09, 0C), a two-word item is placed on the Declaration List (DECLLIST). For REFs, DEFs, and DSECTs this is the eight-character name of the entity; for CSECTs it is a dummy (zero) item. The items on this list, placed there by GENDEF, GENREF, GENCSECT, and GENDSECT, are in one-to-one correspondence with the declaration numbers referenced by object code expressions and load items. In addition to DECLLIST, there is a three-word table (DECNOTBL) whose entries yield the declaration numbers corresponding to section biases: P-section, V-section, and the blank COMMON section. This table is set up by GENCSECT and GENDSECT. When the declaration number of a particular entity must be found (in order to gen an expression or load item), Find Declaration (FINDDECL) is called. It looks at the Relocation coode (see 2.28.2); if that is between 1 and 3, FINDDECL simply returns the corresponding DECNOTBL entry. If the Relocation code is between 4 and 6, then FINDDECL searches for the given name (in IDCENT) in the Declaration List; if found, then the DECLLIST pointer is converted to the corresponding declaration number (this is done by a short machine language subroutine, GDECNOML); if not found, FINDDECL returns with the Answer FALSE, indicating "this entity has not yet been declared".

The Object Code Expression generator (GENEXP) requires a value and a Relocation code (see 2.28.2). If the value is nonzero, it multiplies it by four (to shift from Word to Byte resolution) and gens an Add Constant item (01) for it. Then, if the Relocation code is nonzero, it calls FINDDECL and gens a 20 item (Add Declaration Value at Byte Resolution). Finally, it outputs Expression end (02). There is an alternate entrance (GENEXPP0) for generating an expression to location zero in the P-section: it sets the value to zero, and the Relocation code to one, and falls into GENEXP.

2.29 Object Listing

An object listing consists of three parts, the prologue, the body, and the epilogue. The body is by far the most important, and in fact many FORTRAN compilers have little if any prologue or epilogue. These parts will be described in the subsections following. The summary might be thought of as part of the object listing too, but since it appears independently of the LO option, it is described in a separate section. The prologue and epilogue are both printed by the assembly PASS3F, while the body is printed by OUTPROG.

A great deal of thought went into the design of the object listing, in order to make it as informative and readable as possible. The object listing, like diagnostics, is an area where excellence may be of marginal necessity (most FORTRAN compilers seem to treat it this way), but where a lot of nice things can be done to make life easier for the user.

There are two approaches commonly used to produce object listings, neither of which is completely acceptable. The first is essentially to take the generated code (in hexadecimal or whatever) and try to reconstruct some sort of symbolic assembly language version for it. This is unlikely to produce anything symbolically related to what the user wrote, as can be seen by examining an object listing produced by, say, the IBM 360 H level FORTRAN compiler. It is difficult to keep track of the proper names for scalars and arrays, not to mention dummy identifiers. The "lister" may know only the location and hexadecimal value of a constant, and have no idea whether the constant should be shown as integer, real, Hollerith, hexadecimal, or whatever. Branch locations may show only internally created labels that bear no relation to the labels used in the source program.

To produce a proper object listing, the compiler has to make specific efforts to remember symbolic information, so that it can be listed. A second approach, therefore, is to have the compiler generate assembly language, instead of final object code, and let the standard assembler perform the final assembly pass and make the listing. This is better, but it has a number of drawbacks due to the differences between most assemblers and the FORTRAN language. Using Sigma Meta-Symbol as an example, some of these drawbacks would be:

1. Some of FORTRAN's constants are not allowed; for example, 2.5 (floating point), (1.2,3.4) (complex), 2HAB (left-justified Hollerith), .TRUE.

(logical). These would have to be shown in forms that would be less meaningful to the FORTRAN user.

2. No multiple word literals are allowed. Thus an instruction of the form

```
LD,8      =4.300
```

could not be shown at all. The constant would have to be shown as being addressed remotely with a label, and this simply is not what the compiler does. Complex and long alphanumeric string constants would have the same problem. This would get especially confusing when constants occupy the same storage. For example, a complex and a double precision constant might happen to have the same value, but to show them with the same operand (some label) would be extremely misleading.

3. FORTRAN allows multiple use of the same symbols (because, through context, it knows which are which), and an assembler would have a difficult time doing this. For example: statement function dummies can have the same name as other identifiers; a labeled COMMON block can have the same name as other identifiers; an internal FUNCTION has two entities with the same name, the result scalar, which is used within the funcition, and the entry point, which is used outside the function; calling sequences use special parameters such as P and E in the register field, but they bear no relation to the identifiers P and E that the user might have defined in his program.

4. The compiler wants to print headings in front of some of the sections, such as constants, local storage, etc. Some of these could be handled to some extent with comment cards, but not the constants, because the assembler prints these following the end of the program.

5. The compiler wants to intersperse the user's source lines, number them, and still be able to have them (A) fit on the page (which they would not if they were assembler comment cards) and (B) line up by columns with the generated assembly lines (which they also would not). In other words, for ease of interpretation, column 1 of the generated assembly lines should line up under column 1 of the original source lines. Note: This feature has since been changed (although not the accompanying documentation) so that the columns do not line up. I think that, if at all possible (under the restrictions introduced by XOS), this should be changed back.

6. The compiler splits its generated hexadecimal into its three fields, op-code, register, and address, which we think improves readability.

7. In addition, the tag following the hexadecimal address can be much more informative. It can indicate not only Program, Absolute, and eXternal, but also such things as Variable storage, Common, and Global.

8. The assembly summary could not approach giving the useful, FORTRAN-related, information that the compiler summary gives, so it would somehow have to be suppressed. Furthermore, since the assembler would

most likely be called in after the compiler was finished, the compiler summary would have to be printed ahead of the object listing, which does not please us.

9. There is some question, in any case, whether Pass 2 could directly generate complete assembly language. If not, there would have to be a third pass to do that, and then as long as it is generating the assembly language, it might as well print it too.

It seems clear to us, then, that the compiler must produce its own object listing, and it must be aware that it is going to have to do this, so that it can hold on to the necessary symbolic information. Despite all the above "drawbacks", the object listing is designed to look as much like a Meta-Symbol listing as possible, so that (A) the user does not have to learn a whole new set of conventions, and (B) generated code can be copied from the object listing into an assembly language program with as few changes as possible. Thus the compiler can be useful in showing assembly language users what kind of code they ought to write, especially in interfacing with FORTRAN compiled programs or library routines.

2.29.1 Prologue

The prologue is printed by the assembly PASS3F, and consists of only three things: CSECT 1 (or CSECT 0 if NMP is requested); SYSTEM SIG5F (or SYSTEM SIG7FDP if S in column 1 has been requested, because the user may have used non-Sigma 5 instructions); and SYSTEM FORTLIB (which, for a machine language user, would be needed to define calling sequence op-codes and tags).

Originally we thought we might be able to print SYSTEM SIG5, if no floating point instructions were generated, but this checking was never added to Pass 2. It probably could be.

2.29.2 Epilogue FINISHLO

The epilogue is also printed by PASS3F, following the body of the object listing (described below) and preceding the summary listing (described in Section 2.30). Like the prologue, it contains information that often is shown on other compilers' object listings, such as where the constants, temps, and dummies are, or (and this is often very hard to determine on other systems) how blank and labeled COMMON areas are declared in assembly language.

There are six possible sections in an epilogue. Some of them may not be printed if there is nothing in them. All but the END line are preceded by a heading.

1. Constants. These are the literals that must be generated at the end of the program. GENLIT (Section 2.27) has already gathered these from the various constant lists and accumulated them on the LITERLST. We have to

step through them even if there is no LO requested, in order to be sure that they get output into the binary object file. However, if there is no LO, we print neither the heading nor the individual constants as they come along.

If GENLIT determined that the literals must begin on a double-word boundary, we print a BOUND 8 first. Then we step through the literals, one word at a time. We send each to BODATUM for binary output. If LO is in effect, we increment and print the program location counter, print the hexadecimal value (with no gaps between digits, since the op-code/register/address separation does not apply), and the relocation tag "A" for absolute.

2. Local Storage. This includes local variables, dummies, and local, private, and real time temps. If there is no local storage, we skip this whole section. This is determined by the value of VSIZE. ALLOCATE has set up VSIZE to include local variables and dummies. PASS3F has already invremented VSIZE to include the temps.

The local variables are combined into one RES. There is no point in listing them out, since this is done in the summary. Note, however, that this results in one place where the assembly listing is incomplete. An assembler would not know where the individual local variables were. But to put them all in would just waste too much space.

The dummies are listed next, by stepping through the symbol table looking for identifiers with the SOF trait DUMMY. Each is listed with its name, a RES 1, and the word "DUMMY" in the comment field.

Either the normal or private temps can come next, depending on where ALTEMPS (Allocate Temps, at the beginning of PASS3F) decided they should go. Usually the normal temps come first. Either group may have to be preceded by a BOUND 8, if there were double-word temps used. This is determined from the TEMPEVTR (Temp Even Trigger) and PTMPEVTR (Ptemp Even Trigger), which were set up by Temp Maker and Private Temp Maker, in Pass 2. Each temp has its number printed, followed by "TEMP" or "PTMP" (this is its label), then a RES 1, and the words "TEMP" or "PRIVATE TEMP" in the comments field.

The private temps need to be processed even if LO is not requested, because 1PTMP has something special generated in it, and this must be output via BODATUM. Furthermore, in this case, the listing, instead of showing a RES 1, has a DATA 0, and the comment says "DEBUG CHAIN WORD".

If there is a real time temp, as indicated by RTTEMPCT, it also must have data generated in it. The first half word is the re-entrance flag and the second half is the size of local storage. This is printed as

```
1RTMP     DATA,2        1,size        REAL TIME DATA SENTINEL
```

3. COMMON and GLOBAL. If there is no LO, we skip entirely these last four sections. If the sum of BLCOMSZ, and GLOBALSZ is zero, we skip this section. Otherwise, each COMMON and GLOBAL area is printed as a DSECT 0 followed by a RES of the appropriate size. The name is in the label field of the DSECT. For blank COMMON, this is F4:COM, as determined by the Sigma binary language format. The GLOBAL areas are determined by stepping through the symbol table looking for identifiers with the SOF trait GLOBAL. Their size is computed by a PASSOF subroutine, COMPSIZE.

4. External References. These include FORTRAN names and special (usually library) names. The FORTRAN names are found on the symbol table and have the EXTREF trait. This trait is set up by Pass 2, either in Standard Sprog Call Gen (usually) or in Grntee Addressable Own Mode (when the name of a subprogram is passed as an argument but never directly called). The special names are mainly library references that have been generated by Pass 2 and placed on the REFLIST. Special names can also get on the REFLIST in SINCOLM1 (in-line assembly code). There, any non-FORTRAN name is assumed to be an external reference.

   Several references will be printed on the same REF line. We keep track of the print character position (PCHPOS) to ascertain whethere ther is room enough to print another name. If the current position is beyond ACOLZERO (assembly column zero) plus 72 minus 8, we start a new REF line. This test has recently been incorrectly changed to accomodate on-line listings that can only go out to column 44. The current test neglects to subtract the 8 (to allow for the upcoming name) and therefore almost always requires an extra line on the paper, which was the whole idea this test was supposed to prevent. In any case, the restriction to 44 columns should only be done if we are actually on-line.

5. External Definitions. There are two possible special names here, as well as any number of FORTRAN names. The FORTRAN names are symbols that have both the traits SPROGDEF and EXTERNAL (set up in Pass 1). They are each printed on a separate DEF line, with their definition location shown at the left in the location counter field. This can be determined from the YALODISP rung of the Symbol Table entry.

   The special names are "4MAIN", if the program is a main program, and V#name, if there is any local storage. 4MAIN is printed as the very first DEF, with a location counter of zero.

   V#name is rather unusual. It is the only symbol associated with the FORTRAN system in any way that can be more than eight characters long, and this required some finagling when it was added. It is constructed from the subprogram name (or "4MAIN") preceded by "V#". It is printed with a location counter of zero also, but with a relocation tag of V rather than P. There are a couple of problems currently with the way it is handled. First, it ought to be printed after the FORTRAN names, so that the first name DEFed will either be 4MAIN or the name of the subprogram. Second, in order to conform to proper assembly language format, the V#name should not be DEFed without being defined anywhere. In

particular, the V#name should be defined in the label field of the CSECT that declares the local storage area.

6. End line. This consists of the op-code "END" followed, if this is a main program, by the operand 4MAIN, to declare the transfer address.

2.29.3 Body Interface Items

The body of the object program is printed in the assembly OUTPROG. It consists of a simple, short loop that picks up interface items from Pass 2 one by one, and decides how to print and/or process each one. The interface items fall into five classes, as shown below. Each interface item will be identified here by the name of the subroutine that processes it, the name of the P2P3ROOT item value that is passed, and the description of the operation to be performed.

2.29.3.1 Not directly related to printing

1. LOADPLEX (P23PLEXL) -- Load Plex List

   Plexes are passed from Pass 2 to Pass 3 only in the case of address functions used on in-line assembly instructions.

2. OUTDTORG (P23DTORG) -- Output DATA org with addend

   This, and the next two items, are used only for binary output of DATA statements. These values are not shown on the object listing. This is one of the few places (another being local variables) where the object listing does not show everything it knows, because of the possibly enormous amounts of space that could be used up.

3. OUTDCON (P23DCON) -- Output DATA constant

4. OUTRPROG (P23RPROG) -- Output return to "P" (Program) location counter after DATA

2.29.3.2 Set up

1. OUTADND (P23ADND) -- Output addend

   This receives an addend that is to apply to the next instruction, and saves it in the cell ADDEND. If more than one addehd is received, they are added up.

2. OUTCNL (P23CNL) -- Output comment next line

This item is produced by the CNL POP (see Section 2.24). When the address of a comment is received (the comments themselves are down in P2P3ROOT), it is saved in COMNTLOC, to be used on the next instruction.

3. OUTEND (P23END) -- Output end of program

This just sets the trigger ENDTRIG which is tested after each item. This causes OUTPROG to exit to PASS3F.

2.29.3.3 Normal instructions

This section is used for all the instructions except S, SF, LCFI, and certain immediate instructions (which are discussed in the following section). It is not used for items that are values rather than instructions (those are discussed in Section 2.29.3.5). The normal instructions have a great deal in common: they increment the location counter by one and print it; they print the hex value, divided into three fields (op-code, register, and address); they may have a label defined; they have an op-code; they generally have a register field; they may be indirect; they have an operand, of certain permissible kinds; the operand may have an addend; they may be indexed; they may have a comment. The only place in which they differ is in how the op-code gets printed: we have to list calling sequence op-codes differently, because they look exactly like other instructions in hexadecimal; we do not want to list all branch instructions as BCR,n or BCS,n, so we have to look at the register field to determine how these should be printed; we want to list BEZ differently from BE, even though they are exactly the same instruction (Pass 2 knows which it wanted to generate and sends a different kind of interface item so that they can be listed properly); and library links require a little special handling because they are not passed exactly like the other instructions. These differences will be discussed below. For the most part, what it amounts to is that each type of instruction provides a place to call to get the op-code printed, then they all fall into OUTNORM (Output Normal Instruction), which we will look at now.

Most interface items are two-word quantities, the first word being the item indicator and the second word the operand pointer. We call UNPAKNML, a machine language subroutine, to unpack the indicator into: an R-field listing mode (RFMODE); an immediate trigger (IMEDTRIG); an indirect trigger (INDTRIG); an op-code value (OPCODVAL); an R-field value (RVAL); a combined op and R value (OPRVAL), used by branch instructions; an index value (XVAL); and a preliminary hex word (HEXWORD), which is complete except for the address field, which will come from the operand pointer.

Then we call EVALPNTR (Evaluate Pointer) to decode the operand pointer into a displacement relative to something. The displacement is returned in W0, with a code in Z1 that indicates what it is relative to. If it is relative to a labeled COMMON block, GLOBAL symbol, or external reference, the appropriate symbol name is returned in IDCENT. To the displacement we add our addend, if any. Then we mask this to 17 bits, unless the instruction is immediate, in which case we mask it to 20 bits. This address gets merged into the HEXWORD, which is then ready for both binary output and printing.

Next we call PRLOCHEX, which guarantees object mode (see below), prints the current "P" location, prints the hex value in three parts, and prints the label field of the assembly line, if there is one. To guarantee object mode means to upspace one line if the last line printed was a source line rather than an object line (see OUTSOURC under Section 2.29.3.5). The label can be a symbol (e.g., an entry point), a local label (followed by S), a nonlocal label (followed by S), or a created label (followed by G).

Then, to print the op-code, we call via P0 the routine provided by the particular kind of instruction being processed (see subsections below). The value RFMODE determines which of three ways to list the R (Register) field: 0 to 15; -8 to 7; or with P, C, and E tags for calling sequences.

If the INDTRIG is set, an asterisk is printed one column ahead of the operand column, so that it will stand out to the left. Then we call PROPRNDX to print the operand, addend, and index fields (see below). Then PRCOMNT, which if there is a comment waiting around, tabs to the comment field and prints it. Finally, we zero the addend, increment the location counter, and exit.

Printing of the operand will be discussed in subsection 2.29.3.3.1. First let's examine the various ways that op-codes were printed, depending on the interface item that was encountered and thus the routine that called OUTNORM.

1. OUTOPCD (P23OPCD) -- Output normal op-code

   This is used for all instructions except branches, shifts, LCFIs, calling sequence mnemonics, library references, and certain immediate instructions (those that have a constant, but not literal, operand). It addresses OPCDTB, a table of all the op-codes in order by numeric value. It could, in fact, print a shift or a branch if it had been written on an in-line assembly line as S, SF, BCR, or BCS. When the compiler generates these instructions, however, it never uses these general mnemonics. It uses the specific ones, such as SAD, SFS, BLE, and BGZ, and these are printed elsewhere.

2. OUTBOPC (P23BOPC) -- Output branch op-code

   Used for all the branch mnemonics that are not comparisons with zero, e.g., B and BG, but not BGZ. Instead of OPCODVAL, it useds OPRVAL to address a table of branch mnemonics that are in order by the numeric value of their combined op and R field values. Most of the entries in this table are blank because the compiler only generates a few of the possible condition code combinations.

3. OUTBOPCZ (P23BOPCZ) -- Output branch/Z op-code

   Used for the branch mnemonics that <u>are</u> comparisons with zero. It works exactly like the other branch processor, except that it prints a "Z" after the op mnemonic. Pass 2 has to know which kind of branch it is generating, based in general on whether it has previously generated a compare instruction.

4. OUTOPCS (P23OPCS) -- Output calling sequence op-code

   Used for the calling sequence mnemonics, such as PZE, INTG, etc. Since these mnemonics have the same numeric representation as other op-codes (e.g. LOGL is the same as AI), they have to be specially identified by Pass 2. There are only a few of these, so rather than have a large, empty, table, we use a static list, the OPCSLIST, and search it for the required value and the corresponding mnemonic. If no such value is found, this is an error, so we print "----" in the op-code field.

5. OUTBLC (P23BLC) -- Output library call

   Rather than generate library calls as normal two-word items, with a BAL,6 op-code and a REFLIST operand, Pass 2 sends them as one-word items, containing the address of a library link cell (which is in P2P3ROOT). OUTBLC knows that this means it should create a BAL,6 and submit it to OUTNORM as if it had come through the normal channels. The operand is given to OUTNORM as a pure address, with the upper bits zero. This will be interpreted as a ZEROLIST pointer, and the operand processor (PROPERND) will know that this points to the text name of the library routine and can be printed just as if it were a pointer to the REFLIST or the Symbol Table.

2.29.3.3.1 Print Operand PROPERND

Printing the operand is a key part of producing a readable object listing. It probably requires the most planning to be sure that the necessary symbolic information is availabble at the time the listing is printed. There are 28 different things that an operand can be, and they fall into eight classes. Some of these are very straightforward to get printed (e.g. most identifiers); some are quite complicated (constants have a whole assembly, PRINTCON, just for that purpose); one is actually recursive (an address function may contain an address function)! In each case, the goal is to print something clean and meaningful, and in Meta-Symbol format whenever practical.

2.29.3.3.1.1 Absolute

Indicated by SOT ABSINDIC. If there is an addend, it does not have to be printed separately; it can be added in. The resultant value can then be printed just as if it were an addend, namely as a negative value (minus sign followed by the absolute value) if the 20th bit (X'80000') is set, otherwise as a normal unsigned decimal value. Note that absolute does not apply to instructions such as the LI,9 23 generated for the statement J=23. In this case, the 23 is a constant, not an absolute, and is handled by the special immediate instruction processor (OUTIMMOP). The instruction LI,9 =23, on the other hand, is processed as a normal instruction.

2.29.3.3.1.2 Symbol Table

Indicated by any of the following testing TRUE:

SOT SCALAR

SOT ARRAY

SOT SPROG

SOT MULTDUMY

SOT UNUSED0

All of these can be handled identically, due to the unified structure of the Symbol Table. Note also that dummies are automatically included, because they must fall into one of the above identifier categories. Thus there is no problem in addressing even statement function dummies correctly by name. They all have a Symbol Table entry, and the name is in the first two words. The PAM POP is of course used, in order to suppress trailing blanks.

2.29.3.3.1.3 External reference to non-FORTRAN name

This is indicated either by SOT REFCODE (an entry on the REFLIST, probably placed there by SINCOLM1, the in-line assembly language processor) or SOT ZEROCODE (a library link, in P2P3ROOT, created by OUTBLC, as described above). In either case, the required name is in the first two words pointed to, so these can be treated exactly like Symbol Table names.

2.29.3.3.1.4 Label

Indicated by SOM LABEL. There are three kinds of labels. For each, the numeric value of the label is contained in the YLABLNUM rung of the appropriate list, so it is merely necessary to print this value (with a PDM POP), then follow it by S (for local labels, indicated by SOT LOCLBLCD), or by S (for non-local labels, indicated by SOT NLCLBLCD), or by G (for created, or generated, labels, indicted by SOT CRELBCOD).

2.29.3.3.1.5 Temp

There are also three kinds of temps. The temp number is contained in the address portion of the operand pointer. (Actually, this number is one less, since it begins at zero, whereas we want to print them beginning at one; so we add one to it.) Following this is printed TEMP (for ordinary temps, indicated by SOT TEMPNTR), or PTMP (for private temps, indicated by SOT PTEMPPNT), or RTMP (for real-time temps, indicated by SOT RTTEMPTR).

2.29.3.3.1.6 Address Function

Indicated by SOT ADRFPLEX. This is the only time that plexes are transmitted to Pass 3, and it results only from the use of address functions in in-line assembly code. There are four address functions, BA, HA, WA, and DA. By PULing the plex, we can find out which it is and print, for example, "BA(". Then we recurse through PROPERND to print the suboperand (this can go on indefinitely), then print the closing right parenthesis.

2.29.3.3.1.7 String constant

Indicated by SOT SCONSCOD. Long alphanumeric string constants appear in PAUSE and STOP statements and in calling sequences. If possible, we will print the entire string. For example,

LI,7 ='WHO KNOWS WHAT EVIL LURKS IN THE HEARTS OF MEN'

If it is going to run beyond column 72 of the simulated assembly listing, however, we back up slightly, terminate the string (with a quote), and add "ETC...".

To print a string constant, we pick up the character count from the YSCONSZC rung of the SCONSLST, then step through the text contained in that list, one character at a time until the count is exhausted. We also look at each character to see if it is an apostrophe. If so, we have to print it as two apostrophes since the whole string is enclosed in apostrophes.

2.29.3.3.2 Print Constant Operand PRINTCON

A constant is indicated by SOF CONSTFLG (assuming string constants have already been tested for). A constant on a normal instruction is assumed to be a literal; that is, it will be printed with a preceding equal sign. If a constant is to be used as an immediate operand (e.g. LI,9 23), it must be generated as a special immediate instruction, and will be processed by OUTIMMOP (Section 2.29.3.4).

Printing constants nicely is complicated enough that a whole assembly, PRINTCON, is devoted to that. This is one of the few places where Pass 3 has to reconstruct a symbolic representation for a source program construct, given only the value of the constant (on the appropriate constant list). There are several reasons why we do not just carry along the original symbolic representation from the source program.

A. The user may write his constant in a peculiar form, such as with lots of preceding or trailing zeros, and we would rather print it in a shorter, cleaner form. (The original form might not even fit in the operand field, or even on a whole line.)

B. He may have written the constant with more significant digits than we are actually able to carry. Rather than give the misleading impression that we have maintained that precision, we print it with the proper number of significant digits.

C. The compiler occasionally generates constants, without their having appeared explicitly in the source program. For example, an integer constant appearing in a real expression is converted to a floating point constant and therefore listed as such. In addition, if the compiler does compile-time arithmetic (it currently does some on integers but not on floating point), the resultant value is new.

D. Pass 3 is not the smallest pass, and time is particularly important while printing an object listing, so it is cheaper to make Pass 3 do some extra work than to carry around long-winded symbolic representations of constants.

Since this work is very cheap, a good deal of incremental elegance is built into the process of printing constants on the object listing. Like the allocation diagnostics (Section 3.8.10), this may represent somewhat of an overkill of the problem, but it produces nice results. The various types of constants will be discussed individually.

1. Integer. Integer values are printed as left-justified number with no leading zeros, but with an optional leading minus sign. Zero is printed as 0.

2. Logical. Logical values are printed as one of the following:

   .TRUE.

   .FALSE.

   Not that logical constants are carried on the same list as integers (and may also use the same literal). It is the mode of the pointer that determines how the constant should be listed.

3. Hexadecimal. Hexadecimal values are printed as

   X'hh..hh'

   with leading zeros suppressed (except that zero is X'0'). Not only are the user's hexadecimal constants maintained and listed in hexadecimal, but the compiler occasionally generates constants that ought to be shown in hexadecimal. For example, if the compiler does compile-time arithmetic on the quantity 4HABCD+237, the resultant constant will make no sense in either Hollerith or integer, and so is listed in hexadecimal. Floating point "dirty" zeros are also listed in hexadecimal, but this done by the real/double constant lister (below).

4. Hollerith. This includes both the nH form and the nR form. PRINTCON determines which to use by examining the constant value. If the first byte is a binary zero (which does not represent a printable character), the R form is used, with 0 n 3 and leading zeros suppressed. Note that a value of zero would therefore be printed as "0R".

If the first character is not a binary zero, the H form is used, with 1 n 4 and trailing blanks suppressed. However, the value consisting of all blanks is listed as "4H '.

5. Complex/Double Complex. These are printed as

   (real,real) and (double,double)

   respectively, corresponding to their format in a source program. The real and double values are listed as shown below.

6. Real/Double Precision. As with G format output at run-time, floating point numbers are output in either of two forms, depending on their values.

F-form	n.m	or	n.mDO
E-form	n.mEe	or	n.mDe

However, we wanted to go a little farther than what G format does in providing easy-to-read values. For example, G format will print .123456 in just that form (F), but will insist on switching to E form on the value .002, even though it comes out even after three digits and is thus shorter and more readable than 2.0E-3. However, .000000000000200 is probably less readable than 2.0D-13, because there are too many leading zeros. So we make the rule that if a value comes out even within the proper number of significant digits (6 for real, 15 for double), we will print it in F-form, unless it has more leading zeros than a certain parameter, which is currently five.

Similarly, too many digits to the left of the decimal point make the value hard to read. For example, we feel that 27900000000000.000 is less readable than 2.79D13. So there is another parameter (also currently five) that determines the maximum number of digits that will be printed to the left of the decimal point in the F-form.

Finally, instead of beginning after the decimal point on the E-form (for example .279E14), we use "standard scientific notation" and put one digit to the left of the point (for example 2.79E13). This is as if there were a 1P scale factor in effect.

Thus the following examples can be given:

Real value	Output
0	0.0
-5	-5.0
1/3	.333333

.0000123	1.23E-5
.000012	.000012
.000001	.000001
.0000001	1.0E-7
32100	32100.0
321000	3.21E5

Double precision value	Output
0	0.0D0
-5	-5.0D0
1/3	.333333333333333D0
.0000012345123456	1.2345123456D-6
.000001234512345	.000001234512345D0
.00000012345	1.2345D-7
32100	32100.0D0
321000	3.21D5

This can all be expressed algebraically, in terms of four assembly-time parameters, shown below with their current values:

NS = 6 = number of significant digits in single precision

ND = 15 = number of significant digits in double precision

ML = 5 = maximum number of digits to left of point in F-form

MR = 5 = maximum number of insignificant (leading) zeros to right of point in fractional F-form

Then the form used depends on the value, x, as shown below:

Condition		Form
$\lvert x\rvert \geq 10^{ML}$		E
$1 \leq \lvert x\rvert < 10^{ML}$		F
$10^{-MR} \leq \lvert x\rvert < 1$	and x can be given exactly in at most NS digits	F
(same)	but x cannot be given exactly in at most NS digits	E
$0 < \lvert x\rvert < 10^{-MR}$		E
x = 0		0.0

For double precision, the procedure is the same except ND is used instead of NS, and the F-form values have DO after them.

There is one further exception. If the value has a fractional part of zero, but a non-zero exponent (the so-called "dirty" zero generated by the AINT function), it is listed in hexadecimal, eight digits if single precision, sixteen if double. It is impossible to list such a number in floating point and show the unusual hexadecimal exponent.

2.29.3.4 Abnormal instructions

1. OUTIMMOP (P23IMMOP) -- Output immediate instruction with constant operand

   This is used to distinguish between instructions such as

   LI,9 23 and LI,9 =23

   The latter is handled as a normal instruction. The former uses the same op-code, and has an operand that points to the same constant on the ICONSLST, so it has to be distinguished by processing it in a special place, namely here. This form is the result of a BIM POP, whereas other immediate instructions may be built by the BIN or BIF POPs.

   OUTIMMOP duplicates most of the work that OUTNORM does, except that: it need only print normal op-codes (like OUTOPCD); it cannot be indirectly addressed; the operand (which is a constant) is printed without the preceding equal sign.

2. OUTFABS (P23FABS) -- Output full-word absolute instruction

   Used only for the following three groups of instructions:

   LCFI (i.e. NOP, LFI, LCI, LCFI)

   SF (i.e. SFS,SFL)

   S (i.e. SLS,SLD,SCS,SCD,SAS,SAD)

   These instructions have only absolute numeric values as their operands (i.e. no labels or symbols or temps or floating point constants, etc.), but they locate those operands in strange and mysterious places within the 32-bit word, mainly in various small fields near the low-order end of the word.

   OUTFABS first determines which instruction it has. It addresses another table to print the mnemonic. Then it unpacks the instruction into the appropriate fields and prints them.

2.29.3.5 Other not instructions

These interface items cause something to be printed, but it is not an instruction, as such.

1. OUTNABS (P23NABS) -- Output numeric absolute

   This prints the op-code DATA and a signed decimal integer value for an operand. It is used only in the generation of NAMELISTs.

2. OUTHABS (P23HABS) -- Output Hollerith multiple-word absolute

   The HABS indicator word has a word count in the address field. OUTHABS prints this number of lines, each with the TEXT op-code and a four character text field operand. Since the operand is listed enclosed in apostrophes, we have to look at every character to see if it is an apostrophe; if it is, we print two of them. If there was a statement label, it is listed on the first of the TEXT lines. This is used by FORMAT statements and by NAMELISTs.

3. OUTDBGEN (P23DBGEN) -- Output Debug GEN

   This outputs a line of the form:

   ```
   GEN,1,14,17   x,y,z
   ```

   where x and y are passed along in the indicator word as absolute values, while z is the real operand and is printed in the same way as operands for normal instructions. This constrcut is used in run-time Debug mode (DEBUG option) for check-in data.

4. OUTDEF (P23DEF) -- Output label definition

   This may or may not actually print something. Normally, when a label definition comes along, we just save it to be printed in the label field of the next instruction. However, if more than one label needs to be attached to the same line (generated labels can do this), we do not try to print them all on the same line. When a second label comes along, we print the first one, followed by EQU $, and then save the second one. This process can continue as long as necessary. Note that we use the same label printing routine here as on a normal line, and that labels in this context need not be just entities from a label list; identifiers can also be defined in the label field (entry points).

5. OUTSOURC (P23SOURC) -- Output source line

   This is used to intersperse the source lines in the object listing. These lines are sent along as text strings from Pass 1, via Pass 2, but only if LO was requested. Because we think it improves readability, we always print a blank line whenever we change from printing object lines to source lines, or vice versa. Thus the source lines are "double spaced" away from

the object lines, but not from each other. This is handled by the trigger OBJTRIG and the two routines GTOBJECT (Guarantee Object Mode) and GTSOURCE (Guarantee Source Mode).

The indicator word contains a word count of the source text string. We pull this many words off the Input List and pile them onto the (previously reserved) WORKLIST, so that we can print them all at once with a PAC POP. We cannot print them directly from the Input List, because if previous passes overflowed and dumped their output to an external file, all of the words we need may not be in memory yet. We can call for them one at a time, and when needed the lower level routine will read some more, but we cannot depend upon their being there all at once.

2.30 Summary Listing SUMMARY

This is essentially the last thing done in the compilation process. Anything done after this (in the assembly FORTRAN4) is just getting ready for another compilation. SUMMARY is BRUed to by PASS3F, so when SUMMARY exits, it returns directly to FORTRAN4. The coding in SUMMARY is very straightforward, once you understand the various printing POPs (see Print POPs 1.14). The comments in the listing are quite adequate to understand most of the logic. We will talk here only about places where it is not obvious why certain things work the way they do. SUMMARY falls into six parts:

1. Initialize
2. Alphabetic Symbol Table Sort (ASTSORT)
3. Output Symbol Table (OST)
4. Output Label Table (OLT)
5. Output Functional Groups (OFG)
6. Output Final Statistics (OFS)

These are discussed below.

2.30.1 Initialize

The PS option takes precedence over the NS option. If PS is requested, you always get the partial summary, even if you also requested NS. Actually, requesting both options can be useful, because NS turns off the few lines that get printed from the source listing when no LS is requested.

If either a source or object listing has been printed, we want to page eject to get the summary on a new page. If neither of those listings was printed, then we can go ahead and put the summary on the current page, with three blank lines separating it from the selected LS lines (if they were printed).

If we are printing the LO listings on a typewriter, we want to cut down the width of the tables, so that they will fit on the limited width paper. Normally the symbol table has three columns; on a typewriter it has two. The label table normally has six columns; on a typewriter it has four. The other tables (functional groups) have either the same number of columns as the label table or two more, depending on whether or not they have to print numbers after the names.

2.30.2 Alphabetic Symbol Table Sort (ASTSORT)

All this routine does is execute a SORT POP on the Symbol Table. The reason it is a separate routine is that, if the compiler had to operate on a character set which did not collate sensibly when sorted numerically, this routine would have to see to it that the sort worked properly. This was the case on the 9300, where the BCD character set did not collate in a nice order. This routine had to go through the entire symbol table, character by character, and translate each character to something else which would sort properly. Then it did the sort, then it went through and put all the characters back the way they were so they would print correctly. Fortunately, the EBCDIC character set does not require this, but if we had to operate with some other character set (for some reason) this would be necessary again.

2.30.3 Output Symbol Table (OST)

If there are no symbols, we get out. Otherwise, we print a heading and step through the symbol table (which has been sorted), printing the name of each symbol, and if appropriate, its type, class (e.g. scalar, subprogram), location, and size. If a symbol has the class ILLEGAL, we ignore it. These are extra unneeded names that were created by the scan of Statement Function Definitions. Dummies receive special handling. A dummy's location has an asterisk printed in front of it (indicating that it is addressed indirectly). Also, instead of a size, the word DUMMY is printed.

The various classes of identifiers are distinguished by their class (or SOT) traits (see Traits, 1.13.1). The trait SCALAR is set up by Pass 1, usually in Expression Scan, but it can be set up wherever a name can first appear and need to be classified as a scalar. At the end of Pass 1 (in Final Pass Through Symbol Table), any names that have not been specifically classified (such as names that have appeared only in a COMMON statement) are registered as scalars. The trait ARRAY is set up only by Allow Dimensions, used mainly by the DIMENSION statement (in Pass 1). The trait SPROG (subprogram) is set up in several places in Pass 1. The EXTERNAL and CALL statements do it, as do the subprogram definition statements, FUNCTION, SUBROUTINE, ENTRY, and statement function definition. Most names become subprograms, however, by being used in experssions as function references (e.g. Y=F(X)). The UNUSED trait is present in the initial words that are put into any new Symbol Table entry (see Identifier Scan in Pass 1), and it remains there until overridden by something else. The MULTDMY (multiple dummy) trait is set up only by the FUNCTION, SUBROUTINE, and ENTRY statements.

DUMMY is a SOF trait rather than a SOT trait; it is an attribute that can apply to many different classes of identifiers. That is, a name can be both a dummy and an array (or a scalar or unused, etc.). The DUMMY trait is set up only by appearing in a dummy list in a FUNCTION, SUBROUTINE, ENTRY, or statement function definition statement. In general, the SOT (class) trait that goes with a dummy is not known until succeeding statements.

A scalar always has a size (in elements) of one. For arrays, the Symbol Table entry contains a pointer to the Array List entry, which contains a pointer to the Integer Constant List (the size cannot be a complicated expression, because only dummies have that, and they are handled separately), which contains the value to be printed as the size. Subprograms have a location printed only if they are defined within the program (indicated by the SPROGDEF trait). If they are intrinsic, INTRIN is printed. Otherwise, they are assumed to be externally defined, so EXTERN is printed.

A name can have the class UNUSED if it has appeared in a type statement or a dummy list but has never been actually referenced or allocated. Most unused names have only their class (UNUSED) printed, no location or size. But here again, dummies are handled differently, and do have a location (in temp storage) and have DUMMY printed for their size. Multiple dummy is a special class, and has MULTD printed for its class, and is then handled like other dummies.

2.30.4 Output Label Table (OLT)

This table lists all the local labels first, followed by all the non-local labels. The local labels may be divided up into several files because of END LABELS statements or internal subprograms. We do not care about that any more; we want to sort them all together. This means there may be several labels with the same value (but not the same location). If there are no local or non-local labels, we do not print the table. Otherwise, we print either four or six columns, depending on whether we are listing on the typewriter or not. The label value is contained in the low 20 bits of the first word in the label list entry. (The upper bits contain label traits, which we discard. The SORT POP does not want them either, but it is set up with special coding to take care of that. The two label lists specify (in their code word) a key size of zero. Actually, their key size is one, but the SORT and SER POPs know that zero means one, but only the lower twenty bits are significant.) If the label is defined, it has a location to be printed; if not, we print UNDEF.

Note that the labels come out in numerical order of their values, not their locations.

2.30.5 Output Functional Groups (OFG)

There are eight groups of identifier types that have to be individually printed: local variables, blank COMMON, labeled COMMON, global variables, entry points, local subprograms, intrinsic subprograms, and external subprograms required. The first two have their table heading printed whether

or not there are any such variables, and the heading gives the size of the area. Labeled COMMON has a heading for each separate COMMON block. The others have headings only if there are going to be any entries, and the heading does not give size. Most of the tables are printed with names and locations, and in numerical order by location. This is handled by Process ID Group With Locations (PRIDGRL). Three of the groups, global variables, intrinsic subprograms, and external subprograms, have just names, no locations, and the names are printed in alphabetic order. This is handled by Process ID Group No Locations (PRIDGRNL). Each of the groups ADRs a place ot go to test whether a particular name is in that group, and then calls either PRIDGRL or PRIDGRNL. These routines are similar, in that they step through the symbol table, picking up each identifier and calling the place specified to see if it is an "interesting" identifer. If it is, the "place specified" returns an answer TRUE. Also, if a heading has not been printed unconditionally, then the first time that the "place specified" finds an interesting identifier, it prints an appropriate heading. When printing names and locations (PRIDGRL), the entries are placed into numeric order by location as follows: for each interesting identifier, a two-word entry is created on the SUMRYLST, consisting of the location and a pointer to the name. Then the SUMRYLST is SORTed on the location word.

Following are some comments about how each of the "places specified" determines whether it has an interesting identifier. Local variables are identified by the LOCALVAR trait, set up in ALLOCATE. Blank COMMON variables have the BLANKCOM trait, set up by the COMMON statement in Pass 1. Labeled COMMON variables are identified by seeing whether the fourth word in their Symbol Table entry is a pointer to the particular labeled COMMON block in question (i.e. a pointer to a certain position on the BLNAMLST). No other type of identifier will have a pointer to the BLNAMLST in its fourth word. This pointer is put there by ALLOCATE. Global variables have the trait GLOBAL, set up by the GLOBAL statement in Pass 1. Entry points must have the trait SPROGDEF (set up by the FUNCTION, SUBROUTINE, ENTRY, and statement function definition statements in Pass 1) and also the trait EXTERNAL (set up by the EXTERNAL statement or automatically by FUNCTION, SUBROUTINE, or ENTRY, all in Pass 1). Local subprograms also have the trait SPROGDEF, but do not have the EXTERNAL trait. Intrinsic subprograms have the INTRNSIC trait, set up by Process If Intrinsic (PRIFINTR) in Expression Scan in Pass 1. And finally, external subprograms have the trait EXTREF, which is set up by Pass 2 whenever it generates a reference to a subprogram that has neither of the traits SPROGDEF or DUMMY. In other words, a subprogram does not have to be externally referenced if it is defined within the current program or is a dummy. We could get by without an EXTREF trait at all (Pass 3 could test for SPROG but not SPROGDEF or DUMMY) except for one not very important nicety that we do for the user. If a name is declared EXTERNAL, but never actually used, it will have precisely those traits, but it is a shame to generate (and list) it as a required external subprogram (which will then have to be loaded) when it is not really needed. The EXTREF trait gets around this; Pass 2 knows when it actually generates a reference to something.

2.30.6 Output Final Statistics (OFS)

If the total size of the generated code, constants, local variables, temps, blank COMMON, labeled COMMON, and global variables exceeds 128K, then we might as well tell the user now; our error message will be more meaningful than whatever comes out of the loader. Note that the absence of this message does not by any means imply that the program will load. This test does not take into account other compilations, library routine, Monitor size, actual machine size, or space set aside for foreground or whatever. We would have put this message out at the very end (after the program size table) except that, being an error message, we feel it ought to come out ahead of the line telling how many error messages there were.

Next, if there were any cards with X in column 1, we print the number of them (tallied by Pass 1 in NRXCRDS) and whether they have been compiled or ignored (indicated by XTRIG, set up in FOPTIONS). If there were any error messages, we print that number out, found in NERRMSGS, tallied by PRERR1, the error message printer in Pass 0. If that message was printed (because there was at least one error message) then we always also print the number of statements deleted, even if that count (NRSTDEL, tallied by Release Line in Pass 1) is zero. In any case, we always print the highest error severity found in the program, even if it is zero. Note that the value we keep in ERRLEVEL is not the same as the value we print to the outside world. To conform to the severity conventions elsewhere in the system (see Error Severity in Section 4.5.5), our value is converted according to the following table:

Internal	External
0	0
1	4
2	7
3	10

Level 4 (or 15) errors never get to the Summary; they cause an immediate abort.

Finally we print the little table of program sizes. The sizes for generated code (GCODESZ), constants (GCONSTSZ), and temps (TEMPSIZE) are set up at the beginning of PASS 3. The size for local variables (LOCVARSIZ) is set up in ALLOCATE. Following the line that contains the total of these figures, we test whether there is any blank COMMON, labeled COMMON, or global storage used, and if so we put out all or part of the message indicating that, with some tricky coding to double up on the use of quote words.

The exit from Output Final Statistics is also the exit from Summary and also the exit from Pass 3.

2.31 Abort Errors

The compiler will not abort due to strange or illegal constructs in the source input. It will diagnose any improper input and continue. The only thing that can cause the compiler to abort is a hardware or environment error. These fall into several categories.

1. Bad data. The compiler encounters an item that was previously set up by the compiler itself, but which fits into no acceptable category at the point of discovery. A typical example would be a pointer that does not point to a valid list number or points to a list that could not be used in a given circumstance. The compiler should never create information that will be invalid later. This can only happen because of a hardware error, such as dropping bits or similar actions. Naturally not all incorrect data can be detected, but the compiler often performs checks to verify that things make some sort of sense. For example, if an item can only be one of three things, rather than check for two and then assume it must be the third, it is safer to check for the third and abort if it is not. During initial checkout, there were bugs in the compiler that caused incorrect data to be generated. It is possible, of course, that there still are such bugs, but the likelihood increases with time that the problem is a hardware error.

   Note that the detection of a piece of bad data is only half the battle. It is seldom very straightforward to determine how and when the bad data got where it is. Using the debug package, one can trap on access to any given location (even in a dynamic list), which often leads to the problem area.

2. Memory overflow. When all the available working storage is used up, the compiler is forced to abort.

3. Compiler top over 65K. Because the INT (Interpret) instruction operates on a sixteen-bit address, none of the code in the compiler can be located above location 65535. Otherwise, some of the addressing might not work correctly. (See Section 4.5.6.)

4. Unusual abnormal return from Monitor on read operation.

5. End-of-file or checksum error on scratch file. Actual hardware read errors are handled by the Monitor. The compiler knows when it has received the last piece of scratch information, and it should not encounter an end-of-file prior to that.

6. SI assigned to NO. (See Open Files, 4.4.1).

7. Insufficient scratch file space (RBM only). (See Write a record on a scratch file, 4.2.7.)

The low-level routine that actually performs the abort (M:XXX) to the Monitor is called ABORTML and is described in Chapter 4 (see Abort to Monitor, 4.3.6). A few places (especially in MERRIMAC) call ABORTML directly. Most, however, go through an upper level routine (in POP language) that prints an error message before aborting. There are two such routines.

1. ABORT (in PASSOF)

   This is used by all of the "bad data" abort places in the compiler. It is called in POP language, with two arguments. The first is the address of a 3-word item in the calling program that identifies the name of the calling assembly and its origin in memory. The second is the location of the call and is found on the bottom of the EXITLIST. This enables ABORT to print an error message of the form:

   INEXPLICABLE ERROR AT name+xxxxx. PROBABLE HARDWARE FAILURE.

   before branching to ABORTML. This aids in tracking down the cause of the error.

   Within each of the assemblies that do any checks for bad data, there is a local abort subroutine that is called throughout the assembly. (E.g., in PASS2F it is P2ABORT.) That subroutine sets up the name and origin pointer in P0 (with an ADR POP) and then BRUs to ABORT in order to leave the EXITLIST with the address at which the bad data was detected.

2. ILPRABRT (Interpretive Language Print and Abort, in MERRIMAC)

   This is used only within MERRIMAC on Monitor-related errors, and only when it is safe to enter interpretive mode to print the message. (If the compiler is over 65K or there is no list memory, for example, that is not safe.) ILPRABRT prints the error message addressed by the EATEMP (i.e., by an EAT POP just before the call) and then goes to ABORTML. The error message should be in standard error message format.

Following is a tabulation, by assembly, of all the places that abort.

PASSOF

1. ABORT (described above) branches to ABORTML.

2. OVERFLOW (called by Reassign Memory when working storage is exhausted) prints the error message

   MEMORY OVERFLOW IN PASS n

   and branches to ABORTML.

INTERPTR

1. If working storage is used up, REASSIGN (Reassign Memory) branches to OVERFLOW (in PASSOF), which branches to ABORTML.

2. If there is no working storage initially, INILISTS (Initialize Lists) branches to NOINITMM (in MERRIMAC), which branches to ILPRABRT.

MERRIMAC

1. ABORTML closes some files and does an M:XXX Cal. ABORTML also has the alternate name BPMABORT, used only (curiously enough) by XOS coding.

2. ILPRABRT (described above) branches to ABORTML.

3. If the compiler top is over 65K, GETLMEM (Get List Memory) calls DOWRITE to print a message, substituting ABORTML as the return address (i.e., where DOWRITE will go when it is finished).

4. NOINITMM (No Initial Memory, called by Initialize Lists if it cannot get any working storage) branches to ILPRABRT.

5. If an endOof-file is encountered on a scratch file read, or if there is a checksum error, XREADABT prints the error message

   CHECKSUM ERROR OR EOF READ ON SCRATCH FILE n. PROBABLE HARDWARE FAILURE.

   and branches to ABORTML.

6. If SI is assigned to NO, INITIAL calls DOWRITE to print a message, and substitutes the return address ABORTML.

7. If an error or abnormal return occurs on a scratch file write, under RBM only, DOWRITEX brances to ILPRABRT with an error message.

8. DOREAD can call ILPRABRT, with the error message

   ILLEGAL EOF READ

   in several situations, whose purpose in being added to MERRIMAC I do not understand. These situations seem to be if an abnormal read occurs on a file other than SI or a scratch file; or if an abnormal read occurs on SI and that abnormal condition is not lost data, EOD, or EOF!

STASCAN

Pass 1 does not do much checking of data. It is mainly in the business of creating that data which subsequent passes should check. The only place in Pass 1 where checking is done is in the DATA statement, where all the generation (normally done in Pass 2) is done in Pass 1.

1. If a DATA subscript expression contains a bad pointer, or if an expected pointer to a DATA constant is bad, EXPDPAIR (Expand DATA Pair) calls STAABORT, which branches to ABORT.

PASS2F

1. All the Pass 2 aborts come through P2ABORT, which branches to ABORT. In addition, if it knows what line number it is currently working on, it will the error message

   PASS 2 ERROR ON LINE n

   ahead of the one that ABORT is going to print. If Pass 1 has sent some gibberish to Pass 2, this identifies which statement the gibberish was associated with. Note that Pass 1 sends line numbers to Pass 2 only in run-time debug mode (the run-time debug package needs the line numbers). Therefore, if an inexplicable error occurs in Pass 2, it is often useful to rerun the program with the DEBUG option specified, and hope that the error still occurs. Of course, using the compiler debug package, Pass 1 can also be patched to always send line numbers.

2. PASS2 calls P2ABORT in a variety of situations, including: illegal Pass 1/Pass 2 interface statement indicator number; generated code with unknown instruction class (Pass 2 checking on itself); attempted mode conversion to or from Logical (Pass 1 will not let such a thing get by) or illegal mode value; nonexistent or inappropriate plex pointer for a given operation; attempting to generate an "immediate" instruction on a pointer whose mode is double precision, complex, or double complex.

   These are not, by any means, all the places in Pass 2 where data could be checked. This would become excessive. These places are particularly important and/or easy to check.

ALLOCATE

1. All aborts during allocation come through ALOABORT, which branches to ABORT.

2. The illegal situations checked for are: first LBCOMLST item not a pointer to the BLNAMLST; EQUIVALENCE group does not begin with special code (EQSETBIT); a pointer for a tied variable is not to the EQVCHLST, an item in EQUIVALENCE is neither an array nor a scalar (Pass 1 should never allow such a thing).

PASS3F

1. If a pointeer from Pass 2 has unknown class or allocation, EVALPNTR (evaluate Pointer) calls P3ABORT, which branches to ABORT.

OUTPROG

1. All aborts in OUTPROG come through OPABORT, which branches to ABORT.

2. Situations checked for are: op-code class (from Pass 2) unknown; operand pointer unknown; "immediate" instruction with indirect address; full-word ABS instruction neither shift nor LCFI.

PRINTCON

1. If the pointer provided by OUTPROG is not to one of the constant lists, PRINTCON branches directly to ABORT.

SUMMARY

1. If a symbol table pointer has an improper class or allocation, OST (Output Symbol Table) calls P3ERROR (which is misnamed and should have been called SUMABORT), which branches to ABORT.

CHAPTER 3. THROUGHVIEWS

All of the statements, and a variey of language constructs, are followed through the compiler from Pass 1 to Pass 2 and occasionally into Pass 3 and the Run-Time.

3.1 Identifiers

With two exceptions, all identifiers are kept on the Symbol Table (SYMTABLE). Let's get the two exceptions out of the way first:

1. Labeled COMMON block names are kept on the BLNAMLST (Block Name List). It is probably not absolutely necessary that this be done, but it gets around the problem that these names can be the same as other names, in an easier fashion than some of the other name conflict problems (e.g. see Section 3.76.1.1). The block names will not appear in the printed summary symbol table, but they have special headings all their own in the summary, so they are not lost.

2. Special (non-FORTRAN) identifiers are kept on the REFLIST. These come from two sources. Pass 2 generates some of them as library references (e.g., 9ALOG or 9IODATA). In-line assembly code can reference various kinds of special identifiers that begin with digits or contain certain punctuation marks such as : or #. All of these names are registered on the REFLIST and all are external references. They appear, therefore, in the summary table entitled "External Subprograms Required".

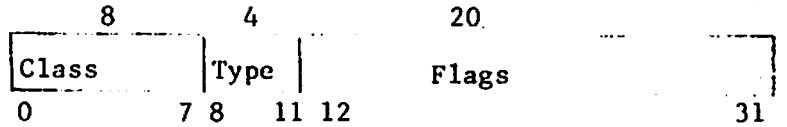
All other names (scalars, arrays, functions, subroutines, intrinsic functions, dummies, multiple dummies) are registered in the SYMTABLE and are printed in alphabetic order in the summary listing. This makes it possible to determine quickly how the compiler has classified any identifier, what its type is, where it is located, and other information, such as the size of arrays.

Each Symbol Table entry consists of six words, as shown below.

1. First four characters of name (left-justified with trailing blanks)

2. Second four characters of name

3. Class, type (mode), and flags (traits). These are all set up at one time or another by the REG POP (see Section 1.7.3.8). This word is arranged:



The class may be:

X'E0' UNUSED (Unclassified)

X'E1' SCALAR

X'E2' ARRAY

X'E4' MULTDUMY (Multiple Dummy)

X'E5' SPROG (Subprogram)

X'EE' ILLEGAL (This is a temporary class, used only by statement functions and DIMENSIONs. See Sections 3.76.1 and 3.37.)

The type values are:

0 UNKNOWN (or no type, such as for SUBROUTINEs)

1 INTEGER

2 REAL

3 DOUBLE

4 COMPLEX

5 KOMPLEX (Double Complex)

6 LOGICAL

The trait flags have the following values. They are individually described in Section 1.13.

	X'80000'	ADRSBLFG	IAddre (Addressable)
	X'40000'	BLANKCOM	(Blank COMMON)
	X'20000'	CONSTFLG	(Constant)
X	X'10000'	CPXMXFLG	(Complex mixture)
	X'08000'	DBLFLAG	(Double in expression)
X'	X'04000'	DATA	(Has appeared in DATA statement)
	X'02000'	DUMMY	
	X'01000'	EQVGLOBL	(EQUIVALENCEd to GLOBAL)
X'	X'00800'	EVENLOC	(Even location)
X	X'00400'	EXTREF	(External reference)
	X'00200'	EXTERNAL	
	X'00100'	GLOBAL	
X'0	X'00080	INTRNSIC	(Intrinsic)

X'00040' IOLUSAFG (I/O list unsubscrpted array)

X' X'00020' LABCOM (Labeled COMMON)

X' X'00010' LOCALVAR (Local variable)

X'00008' NEVALXFG (Not evaluatable in index)

X'00004' SADRFLAG (Signed addressable)

X'00002' SPROGDEF (Subprogram definition)

or

LOOPINDP (Loop-independent)

X'00001' TADRFLAG (Test addressable)

4. Special information. For arrays, this is a pointer to the Array List entry that contains the extra information (offset, size, dimensions, etc.). (See Section 3.37 and ARRAYLST in Section 1.5.)

   For intrinsic functions, this is the intrinsic function number (a code number that identifies the function to Pass 2). (See Section 3.6.1.)

5. During Pass 1, this is the Permissibility word. It starts out as all ones, and is masked down as various things become no longer permissible. Section 2.8.1 describes this and contains a table of the usages permitted by the different bit positions.

   During ALLOCATE, this word can be used as a temporary place to help process EQUIVALENCE chains. It typically contains a pointer to the EQVCHLST (EQUIVALENCE Chain List).

   After allocation, this word contains an allocation pointer that indicates what storage area this identifier belongs in. This can be:

   X'23' BLCOMCOD (Pointer to Blank COMMON List)

   X'24' BLNAMCOD (Pointer to Block Name List, for labeled COMMON)

   X'32' GLNAMCOD (Pointer to Global Name List)

   X'96' PROLOCIN (Special Pass 2/3 operand for "P" program location)

   X'98' LOCVARPT (Special indicator for local variable storage)

(The hexadecimal values shown here, which are the first eight bits of the pointer, should be verified with a current listing of LISTDEFS, as they are heavily parameterized and could change with reassembly.)

6. During Pass 1, this used only for intrinsic functions. It contains the function's type in the mode field. It is also used to determine how the function's arguments should be scanned. Different intrinsic functions permit different numbers and types of arguments, and must diagnose all others. Rather than containing a bit pattern that reflects that, this word contains that address (within EXSCAN) of the subroutine that will correctly process this particular class of arguments. Thus, PRIFINTR (Process If Intrinsic) IRSs through this address to process the arguments. The address is obtained from the intrinsic function's entry in the static list, INTRILST. (See Section 3.6.1 and INTRILST in Section 1.5.)

   Later on, after allocation and Pass 2, this word contains the absolute allocation displacement value that determines the identifier's storage position relative to the storage area that was identified by the previous word.

Whenever an identifier is scanned, the Symbol Table is searched to see if that identifier is already there. If not, it is added, initially with UNUSED class, UNKNOWN type, and no traits. The order of names in the SYMTABLE will therefore be the order in which they appeared in the program. Note, however, that if a statement is deleted (due to a major error), and identifiers that it might have added to the SYMTABLE will be removed when the lists are restored to their original condition.

At the end of Pass 3, when the program has been listed and output, the Symbol Table is SORTed for printing in the summary. Care had to be taken not to sort this too soon. All pointers to the Symbol Table are dependent on their position within the list, and all the positions are changed when the list is sorted. I think, for example, that this is why there is a GLNAMLST (Global Name List). On the 9300, variables that were EQUIVALENCEd to GLOBAL variables had, in their allocation pointer (fifth word in the Symbol Table entry), a pointer to another SYM TABLE entry, namely the GLOBAL variable. This resulted in an interesting problem. When the summary symbol table entry for the equivalenced variabel was printed, we had to determine what storage area it was in, in order to print "G" or "C" or whatever. But with the SYMTABLE sorted, this no longer pointed to the right entry. With all the GLOBAL names on another list, which is not sorted, this problem is alleviated.

For further information, see Sections 2.8 and 2.9.4, also 3.2, 3.4, 3.5, 3.6, and 3.8.

3.2 Arrays

An array is always identified by having been DIMENSIONed. This may happen in a DIMENSION statement, or in another statement such as INTEGER, REAL, GLOBAL, etc., but it must be done before the array is used. DIMENSIONing an array provides two pieces of information. One is the size of the array, which is not really too important until allocation. More important, and potentially

very complicated, are the individual dimension sizes, which produce the subscript multipliers, which are used to compute the location of any given array element. Assume that the array is DIMENSIONed as:

$$A(L_1:U_1 , L_2:U_2 , \ldots , L_N:U_N)$$

and suppose that it is referenced as:

$$A(S_1 , S_2 , \ldots , S_N)$$

The compiler would like to be able to generate code that first manipulated the subscripts in an index register to come up with an index value, X, and then reference the array with an instruction like:

```
LW,8      address,X
```

Since arrays are stored columnwise (first subscript varying most rapidly), each subscript must be multiplied by the product off all the previous dimension sizes. For example, the third subscript (S_3) must be multiplied by both the first dimension range (U_1-L_1+1) and the second dimension range (U_2-L_2+1). If we call the dimension range multipler for each subscript D , then:

$$D_i = U_i - L_i + 1$$

Then the subscript manipulations amount to:

$$S_1 + D_1 (S_2 + D_2(S_3 + D_3(\ldots (S_{N-1} + D_{N-1}(S_N)) \ldots)))$$

This nested structure is easily computed by starting at the inside and working out:

$X_N = S_N$

$X_{k-1} = X_k \ D_{k-1} + S_{k-1}$ for k = N, ... , 2

X_1 = index value

or, in Sigma code:

```
LW,X      S_N
MW,X      D_N-1
AW,X      S_N-1
MW,X      D_N-2
AW,X      S_N-2
```

.

.

MW,X D_1

AW,X S_1

But what is it that this index value gives us? Well, the index value would be zero if all the subscripts were zero, so this index value tells us how far this element is from the zeroth element, which is A(o,o,...,o). However, the array seldom actually begins with the zeroth element. It begins at $A(l_1,l_2,\ldots,l_N)$, i.e. where each subscript is the lower bound for its dimension range. This would be the zeroth element only if, when DIMENSIONed every lower bound was zero.

So what we need is a value that tells us how far the zeroth element is from the actual start of the array (or first element). This value is called the offset, and is computed when the array is DIMENSIONed, at the same time the multipliers are computed. (See "Allow Dimensions", Section 3.37). Thus the actual instruction accessing the array element would look like:

LW,8 A+offset,X

There is one more consideration. All of these formulas give positions in terms of number of elements. Multi-word data types (e.g. DOUBLE PRECISION, COMPLEX) use more than one word per element. Therefore, both the offset and the final index value must be multipled by the element size. So our final formula for array element position is:

$$A + D_0(\text{offset} + S_1 + D_1(S_2 + D_2(S_3 + D_3(\ldots (S_{N-1} + D_{N-1}(S_N)) \ldots))))$$

Where D is the element size for all but dummy arrays with adjustable dimensions, the offset is a constant. The element size is always a constant. So the value of (A+D offset) is typically built into the address field of the accessing instruction. Furthermore, in most cases the Sigma indexing automatically takes care of the element size for the index value, because it doubles the index value when double-word instructions are used. (A double complex array, with a four-word element size, has to be handled specially.) So the element size multiplier is ignored and the access instruction looks like:

LW,8 A+D offset,X

where X is computed as shown above.

To do all of this requires that a fair amount of information be kept around for each array, to be used when generating references to it. This includes the offset and all of the multipliers. It also includes the array's location and its size, which is needed when an entire array (unsubscripted) appears in an input/output list. This information is kept in several places. There obviously is not room for it all in a six-word Symbol Table entry, so the rest is kept in one entry on the Array List and one or three entries on the Group List. This is shown in the following table.

Table 3.2.1 Symbol Table Structure of Array Names

SYMTABLE

NAME	1st half
NAME	2nd half
ID	class (ARRAY), TYPE, AND TRAITS
EXTRA	pointer to ARRAYLST
PERM/ALLOC	Permissibility (in Pass 1) Allocation pointer (after ALLOCATE)
DISP	Allocation displacement (after ALLOCATE)

ARRAYLST

OFFSET	pointer to ICONSLST or PLEXLIST
SIZE	" " "
MULTIPLIERS	pointer to GROUPLST (Group 1)
MESSY MULT EXPS	pointer to GROUPLST (Group 2)
PLEXLIST COPY	pointer to GROUPLST (Group 3)
OFFSET TEMP	PTEMP pseudo-pointer

GROUPLST (Group 1) Multipliers

ELEMENT SIZE	pointer to ICONSLST value of 1, 2 or 4
$U_1 - L_1 + 1$	pointer to ICONSLST, SYMTABLE, or PTEMP pseudo-pointer
$U_2 - L_2 + 1$	" " " " "
⋮	⋮
$U_{N-1} - L_{N-1} + 1$	" " " " "

GROUPLST (Group 2)

1st MESSY MULT EXP	pointer to PLEXLIST
1st PTEMP	PTEMP pseudo-pointer
2nd MESSY MULT EXP	pointer to PLEXLIST
2nd PTEMP	PTEMP pseudo-pointer
⋮	⋮
Kth MESSY MULT EXP	pointer to PLEXLIST
Kth PTEMP	PTEMP pseudo-pointer

GROUPLST (Group 3)

Plexes	Copy of PLEXLIST used during creation of array information

The Symbol Table contains (after ALLOCATE has finished) the array's location, expressed XXX as a displacement from one of the storage areas (e.g. local, blank COMMON, labeled COMMON, etc.). The fourth word is a pointer to the entry on the Array List containing additional information. (See also Section 1.5, under SYMTABLE.)

On the ARRAYLST, the first word is the offset. Usually this is a constant (i.e. a pointer to the ICONSLST), but on adjustably dimensioned dummy arrays, it can be a more complicated expression, in which case this is a plex pointer. If this is the case, we are going to want to compute the offset only once at run-time (at the head of the subprogram) and store it in a temp for use during the rest of the program. So we set aside a private temp for it. This will be pointed to by the sixth word in the ARRAYLST entry.

The second word is the total size of the array. It too may have to be a complicated expression pointer, but it usually is not. The third word is a pointer to the GROUPLST entry that contains the multipliers (D through D).

For non-dummy arrays, these three words are all there is. For dummy arrays, there will be at least a fourth and fifth word. The fourth word is a pointer to the GROUPLST entry containing the "messy" multiplier expressions. This will be explained shortly. The fifth word is a pointer to the GROUPLST entry that contains a copy of the PLEXLIST that was built up while generating these various multiplier and offset expressions.

Later processing knows whether the fourth and fifth words exist by whether the array has the SOT trait DUMMY. In addition, if the offset pointer (first word in ARRAYLST entry is not to the INCONSLST (i.e. the offset is complicated), then there will be a sixth word in the ARRAYLST entry, and that word is the private temp (PTEMP) that the offset will be stored into. In Pass 2, the prologue code generated will then compute the offset and store it in the temp. Then, statements that need to use the offset will get it from the temp. (See also Section 1.5, under ARRAYLST.)

There is always at least one group created on the GROUPLST for each array. That is the N multipliers (D through D) where D is the element size, and is always a pointer to the ICONSLST. The other multipliers are usually constant too, but in the case of adjustably dimensioned dummy arrays, they can be expressions of varying complexity. If the final multiplier is directly addressable in integer, we can always just use it directly. For example, in

DIMENSION A(3:10, J , 0:J , 6)

the three multipliers are:

$S_1 = U_1 - L_1 + 1 = 10 - 3 + 1 = 8$ (an integer constant)

$D_2 = U_2 - L_2 + 1 = J - 1 + 1 = J$ (a simple scalar)

$D_3 = U_3 - L_3 + 1 = J - 0 + 1 = J+1$ (an expression

The first two are directly addressable (the first is a pointer to the ICONSLST, the second to the SYMTABLE), while the third is an expression and must be computed. In this latter case, the ARRAYLST does not contain a plex pointer for the expression. We want these multipliers to be something we can pick up and use immediately. So, the ARRAYLST contains a PTEMP pointer identifying the private temp into which this multipler will be stored after it is computed in the prologue code. Note that PTEMP pointers are really pseudo-pointers. They do not really point to a list. (See "Pointers and Pseudo-pointers", Section 1.11.

For each "messy" multiplier that has to be evaluated and stored in a temp, there will be a two-word item in another group on the GROUPLST There may not be any messy multipliers, in which case this group will not exist. If there are some messy multipliers, there will be K of them (where K N), and thus K two-word items in this second group. The ifrst of the two words is the expression plex pointer, and the second is the PTEMP pointer that tells the Pass 2 generated prologue code where to store the value of the expression. Once this is done, later uses of the multiplier can safely use the temp, knowing that the value has been previously computed. (It is not legal to change the adjustable dimensions in a program, which is the only way that the multipliers could change during the program.)

Finally, when all these expersions (for multipliers and offset) were being built up into plex form, the plexes were being automatically placed on the PLEXLIST. Normally, the PLEXLIST is sent along to Pass 2 from time to time. We do not want this to happen to these plexes. We want them to be on call whenever we have to use them (like in the prologue generation). So we place all these plexes into a third group on the GROUPLST pointed to by the fifth word in the ARRAYLST entry (above). Of course, if the array is not a dummy array, theer will be no plexes, and therefore no second or third group on the PLEXLIST. For further information on arrays, see "Allow Dimensions", Section 3.37; "Set Element Size", Section 2.8.3.5; "Output Symbol Table", Section 2.30.3; and "Subscripts", Section 3.3.

3.3 Subscripts

Subscripts are especially interesting, because they are arithmetic expressions on which an amazing amount of manipulation is done. At first, the several subscripts for a particular array reference are scanned (see 3.10.1.3.2)), and pointers to the expressions are placed on the Script Expression List (SCREXLST). As the expressions are scanned, they are required to be of integer, real, or double precision mode, and any deviation is diagnosed. Then the several expressions are combined with the subscript multipliers into single expression by Combine Subscripts (COMBSCRS in PASSOF) as indicated by the formulas in 3.2. The subscript multipliers are part of the data known about the particular array being referenced: they form a group on the GROUPLST. The multipliers are usually constants, but in adjustable dummy arrays they may be variables or expressions.

Combine Subscripts (COMBSCRS) is complicated by the fact that is doing two things besides combining subscripts. It is separating the constant part of the subscript from the non-constant part, and it is identifying loop-independent subscripts--expressions whose only constituents are constants and DO-loop control variables. Such subscripts can be maintanined in index registers across statement boundaries because they are constant for any one iteration of the loop.

In combining the constant parts of subscripts, we make use of all the algebraic properties of integer expressions: the associativity and commutativity of addition and multiplication, and the fact that multiplication is distributive over addition. These facts are expressed in the following identities:

$$(a+b)+c = a+(b+c) \qquad (ab)c = a(bc)$$

$$a+b = b+a \qquad ab = ba$$

$$a(b+c) = ab + ac$$

Thus in the statements

```
REAL A(10,14)
A(J+6,K+7) = Q
```

the right-most dimension 14 is irrelevant, but everything else matters. The effective subscript is J+6+10*(K+7). This is simplified to (J+10*K) + 76.

In constructing the reference to the array in object code, we incorporate the 76 into the generated instruction and evaluate (J+10*K) into an index register to modify the instruction. You might think that the address in the generated instruction would be A+76, but it is not. We must subtract the offset* of the array, which is 11, so we address A+65. (Note that the offset of an array is the difference between the first element and the zeroth element--the element addressed when all subscripts are zero.)

The above is true for addressing non-dummy arrays, but for dummy arrays the offset cannot be built into the instruction. For a dummy array, the temp with

same name as the array contains the address of the zeroth element. So, for the above example, we evaluate (J+10*K)+76 into an index register, then construct an indirect indexed reference to A.

Subscripts used on scalars and arrays in EQUIVALENCE statements are treated differently from the general types of subscripts. These subscripts can only be constants, and they are combined in ALLOCATE by a routine called Next EQUIVALENCE Item (NXEQITEM, 3.8.4.2). At the time these statements are scanned, Pass 1 does not even know whether the number of subscripts is correct. This is so because EQUIVALENCE statements are allowed to precede DIMENSION statements.

Subscripts on Multiple Dummies are treated very much like subscripts for single-dimensioned arrays. Here there is an implied offset of 1, and this is handled by the fact that the temp with same name as the multiple dummy contains the address of the zeroth element of the multiple dummy; it is set up that way by 9SETUPV.

Within single statements and sometimes across statement boundaries, the compiler recognizes common subscripts--that is, it recognizes reoccurances of the same subscript expression and does not recompute them. Two subscripts do not have to be written the same way to be recognized as equivalent. For example, in the statements

```
REAL A(10,14), B(27)
A(J+6,K+7) = B(J+10*K)
```

the compiler recognizes that the effective subscripts for A and B are the same (except for a difference in the constant part). This recognition of common subscripts is greatly aided by a feature of the plex-building POPs. When a POP is about to put a new plex onto the PLEXLIST, it first searches the PLEXLIST to see if there is already another plex that is just the same. If there is one, we do not append a new plex but instead make a pointer to the old plex. Thus two equivalent expressions really look equivalent because the pointers to them are equal. The searching mechanism is used by the FIP, FIC, CIF, and CIC POPs (See 1.7.3.11). The price (in time) of this searching is paid by all users of plexes, although subscripts are the only ones that get any benefit from it. The searching does bestow a concomitant benefit in the saving of space on the PLEXLIST.

Combine Subscripts goes to some additionla trouble (beyond the searching mechanism of the plex POPs) to assure that common subscripts are recognized. It breaks sums and products down in looking for additive constants, and before putting them back together it sorts the terms (using the SORT POP). This puts the terms in an arbitrary order and guarantees that I+J will be recognized as the same as J+I.

3.3.1 Index Register Usage in Generated Object Code

The seven index registers in the Sigma 5/7 hardware are divided into two sets: the X Regs (registers 1, 3, 5, and 7, where X stands for indeX) and the D Regs (registers 2, 4, and 6, where D stands for DO).

The main use (and almost sole use) of index registers is to handle subscripting.

The X-Regs can be used to handle the most general kinds of subscripts for they are capable of doing all kinds of integer arithmetic. This is because of an oddity of the Sigma 5/7 Multiply and Divide instructions: they work as you would want them to for FORTRAN arithmetic of INTEGER type, but only in odd-numbered registers. They do useful, but different, things in even-numbered registers. (See the computer reference manuals for details).

D-Regs are used for handling very simple subscripts over the range of an entire DO loop. Without exception, the DO Registers (D Regs) contain the control variable of a DO loop, or the control variable offset by a constant amount, the constant being computed at compile-time.

There is a conflict over the use of IndexRegister 6, because 6 is also the library link register. (Library routines are linked through register 6, whereas compiled SUBROUTINEs and FUNCTIONs are linked through register 15.) Furthermore, many library routines also destroy the contents of registers 2 and 4. The compiler knows which library routines destroy which registers. With each library link (associated with the BLC POP) there is a word telling which registers are destroyed, and this mechanism takes place automatically through the BLC POP.

In most cases, non-DO-dependent subscripts can be computed in the X Regs. It is the exceptions to this rule that make subscript generation a mess. Two things can cause these exceptions.

1. We may run out of index registers. I.e., subscripting may occur to a depth greater than four.

2. A subscript may require register 9 for evaluation. The easiest case of this to understand is where a FUNCTION is referenced in a subscript. Since the standard register in which to return results is 9, we cannot evaluate this subscript in register 1, 3, 5, or 7. There is a trait called NEVX (Not EValuable in indeX) that is inherited up through the plexes of an expression. The exponentiation (**) operator also possesses the NEVX trait, because it may involve a call on a library routine. Therefore a subscript such as J**2, even though it is done with in-line multiplication, is done in register 9 rather than in the index register where it is needed.

In most cases, subscript code is generated when and where it is needed. For example, if somebody writes

```
X = Y + Z(J*K+L)
```

he will get the code

```
LW,8    Y
LW,1    J
MW,1    K
AW,1    L
FAS,8       Z-1,1
STW,8       X
```

Here we start by generating a reference to Y. Then we wish to generate a reference to Z(J*K+L), and we know that to do so we have to get J*K+L into an index register. (If it were already in an index register, we would know it and would not recompute it.) So we drop whatever we were doing (which in this case was some floating point arithmetic in register 8) and compute our subscript.

Now let us suppose we have a more complicated subscript. Suppose the source statement is

```
X + Y + Z(J*K + IGGLE(L))
```

where IGGLE is a function of INTEGER mode. Now the compiler does some funny things. It notes that it needs register 9 for the subscript, but observes that it cannot have it just now. (The arithmetic going in register 8 might be double precision or something, and anyway, calling IGGLE might clobber register 8). So now we enter a mode called Retroactive Subscript Generation. What is going to happen is that the subscript evaluation code is going to be pushed up to the beginning of the statement. This can happen even though we have generated part of the statement (the LW,8 Y instruction) already. We make a two-word note on the SCRPTLST (Script List) of a pointer to the expression J*K+IGGLE(L) and a pointer to a temp (say 1TEMP) into which it is to be stored. We now behave as if the subscript expression were 1TEMP, which is addressable. Now later, after the whole statement has been generated, we call a subroutine named SCRPTGEN (Retroactive Subscript Gen) which goes through the SCRPTLST and evaluates expressions and stores them in temps. Furthermore, it so moves the generated code around that the subscript code comes out ahead of the statement code. It really does. Therefore, at the time Pass 2 sees the above statement it generates

```
LW,8    Y
LW,1    1TEMP
FAS,8   Z-1,1
```

```
STW,8       X
```

But later on, at statement cleanup timee, Retroactive Subscript Gen comes along and precedes the aforementioned code with

```
LI,14       1
BAL,15      IGGLE
INTG    L
STW,9       2TEMP
LW,9    J
MW,9    K
AW,9    2TEMP
STW,9       1TEMP
```

Now the reader with the very subtle mind can see that this kind of retroactivity can be nested. The term that we use in the compiler is that retroactive subscripts can have "children". In other words, in the middle of a retroactive subscript there may be a retroactive subscript. Retroactive Subscript Gen is prepared for this contingency (indeed, it turns out to be quite easy to handle). You can figure out from the order in which the temps are used what kind of time-order the following code was generated. The source statement is

```
X = Y + Z(J*K + JIGGLE(Y - Z(J*K + IGGLE(L))))
```

and its object code is

```
LI,14       1
BAL,15      IGGLE
INTG    L
STW,9       5TEMP
LW,9    J
MW,9    K
AW,9    5TEMP
STW,9       2TEMP
```

```
LW,8      Y
LW,3      2TEMP
FSS,8          Z-1,3
STW,8          3TEMP
LI,14          1
BAL,15         JIGGLE
SNGL,P         3TEMP
STW,9          4TEMP
LW,9      J
MW,9      K
AW,9      4TEMP
STW,9          1TEMP
LW,8      Y
LW,1      1TEMP
FAS,8          Z-1,1
STW,8          X
```

It can be seen that, because of retroactivity, not only are the temps used in a peculiar order (5, 2, 3, 4, 1), but the index registers are used in a strange order also (3 is used before 1). The code generator for this was almost as difficult to design as the above object code is to read.

3.4 Dummies

One of the most significant things to say about dummies is possibly that ther is not much to say about them. On some other compilers, dummies are treated as separate entities that have to be tested for and handled specially in numerous places. For example, 900 Series FORTRAN II had a special list for fixed (integer) dummies and one for floating dummies. For most purposes, however, a dummy is essentially something else, a scalar or array or function, and can be handled without knowing that it is a dummy. Thus dummy identifiers are present in the Symbol Table along with everything else, they have a standard "class", like SCALAR, ARRAY or SPROG, and they are identified as being dummies only by having the DUMMY trait flag (testable with SOF).

Multiple dummies, on the other hand, are a separate entity and do have to be processed specially. They have the MULTDUMMY class (testable with SOT) and have their subscripting handled somewhat differently from arrays. They are also present in the Symbol Table.

The DUMMY trait (on ordinary dummies) is important in Pass 1 mainly in two places (1) Dummy arrays may be adjustably dimensioned. (2) A previous dummy can appear again as a dummy in an ENTRY statement, even though the permissibility bits say that it cannot. The SOF trait overrides this (in the coding of the ENTRY statement).

Other restrictions, such as not appearing in COMMON or EQUIVALENCE are automatically handled by the permissibility bits.

Pass 2, on the other hand, tests the DUMMY trait fairly often, in order to generate different object code (with indirect bits) for dummies. Also, special prologue code has to be generated for adjustably dimensioned dummy arrays (see Section 3.78.2.1).

In the summary listing, a dummy appears as a scalar or array or whatever, but its location is preceded by an asterisk, which is an appropriate way of signalling "dummy", because the indirect addressing feature (generally used to address dummies) is indicated by an asterisk in the assembly listing.

Note also that the dummies of statement functions may be the same as other identifiers, including the dummies of other statement functions. Thus, in a program beginning

```
F(X) = X ** 2 - 1

G(X) = 2 ** X + 1

X = 0
```

there would be three Symbol Table entries labeled "X", two statement function dummies (each a scalar), and a local variable (also a scalar). All of these appear in the summary listing. There is no confusion in object code, because each statement uses pointers to the proper entry. There is one slight problem that had to be taken care of, however. If a later statement in the above program referenced X, it would have to be the non-dummy X, since the dummies are recognized only within their respective statement functions. However, the Symbol Table is searched in forward order, which would mean that the wrong X will be found. If the non-dummy X had appeared first, everything would be fine. So when it does not, we pretend that it did and create one. We create an identifier with "just born" characteristics, namely UNUSED class, UNKNOWN type, and no traits. If the identifier appears later (as above), any set of characteristics can be superimposed on this structure. If no identifier with the same name appears, the duplicate identifier will nonetheless appear in the symbol table in the summary listing, as an UNUSED identifier. This may be a little puzzling to anyone who scrutinizes symbol table listings, and we would like it better if it did not happen, but it does no harm. Nobody ever constructs a pointer to it.

3.5 Multiple Dummies

Multiple Dummies are much like one-dimensional arrays. Because of their variable length, they are not brought into the subprogram. That is, argument addresses from dummies other than multiple are brought into temps within the subprogram, for quick reference. Since we do not know how many temps to allocate for multiple dummies, we allocate only one temp--which will point to the vector of argument addresses in the calling sequence itself. Individual addresses are plucked out of this vector when they are needed, not at prologue time.

It is almost never legal to use a multiple dummy without a subscript. The only legal such usage in a type statement. An array may be used unsubscripted in an I/O list or an argument list (meaning the whole array), but a multiple dummy may not.

The symbol MULTDUMY is its ID value. It is a separate class of identifier, different from SCALAR, ARRAY, SPROG.

To be classed as a MULTDUMY, an identifier must appear as the last dummy in the dummy list of a SUBROUTINE, FUNCTION or ENTRY statement. It must appear between slashes and must have a count variable appended to it. For example, the statement

```
SUBROUTINE GLOP (A, B, /MULT(NMULT)/)
```

establishes MULT as a multiple dummy, with NMULT as its count variable. (Note that the count variable is not permanently associated with its multiple dummy. In a later ENTRY statement, MULT could appear with a different count variable.) Count variables are usually local variables, but they can be in COMMON, or dummies, or whatever. An interesting usage is

```
FUNCTION KOUNT (/VARS(KOUNT)/)

RETURN
```

where the name of the function is also the variable that receives the count. Here the value of the function is the number of arguments it is called with. It is strange, and perhaps even silly, but it works.

Multiple Dummy elements can be used in expressions and generally wherever array elements can be used. Expression Scan (EXSCAN) recognizes them as MULTDUMYs and requires them to have a single subscript expression. They are built into the same kind of plexes that array elements are--Fully Subscripted Array Plexes (FULSUBPX) and Constant Subscripted Array Plexes (CONSUBPX).

ADRCALC in Pass 2 recognizes Multiple Dummies and goes to some trouble to address them. On the 9300 they were fairly easy to address because indirect

addressing worked to any depth. On the Sigma, we pick up the argument word, manually check it for indirect, and manually do the extra fetch if it is required. Let us contrast the code generated for a dummy array reference with that for a multiple dummy element. Suppose we want to generate a Divide Word instruction. Then for the array element ARRAY(J) we get

```
        LW,1  J

        DW,9 *ARRAY,1
```

and for the multiple dummy element MULDMY(J) we get

```
        LW,1  J

        LW,1 *MULDMY,1

        BGEZ 1G   BRANCH IF ARG NOT INDIRECT

        LW,1 0,1

    1G  DW,9 0,1
```

a comparatively unattractive piece of code. The comment shown on the BGEZ instruction is generated with the CNL POP.

3.6 Intrinsic Functions

3.6.1 Intrinsic Functions Pass 1)

If you are not thoroughly familiar with the meanings of and distinctions between intrinsic and basic external functions, please read those sections in the reference manual first.

The intrinsic functions are all listed on the Intrinsic List, which is a static list (1.4.3) in Expression Scan. For each function, it contains the name, the type, and a key indicating how many arguments there are and what types they may be. This key is actually the address of a routine that will process that number and type of arguments. The first time we see an intrinsic function, we have to search for it on the Intrinsic List. (Of course this means that we search for all other, non-intrinsic, functions too, but only the first time they appear.) Then we can establish its type and the Intrinsic trait in the Symbol Table. Also, in the fourth word of the Symbol Table entry, we put a number that identifies the function to Pass 2. (This is a count beginning at one that Pass 2 uses to go through a branch table.) And in the sixth word of the Symbol Table entry, we put the address of the argument processing routine. Therefore, when the function is seen again, the type is all set, so is the ID number for Pass 2, and to process the arguments, we do the rather unusual action of branching through the Symbol Table.

Both intrinsic and basic external functions affect and are affected by permissibility. If a function cannot be intrinsic (e.g., it was declared EXTERNAL), it may still be basic external, in which case the only thing we need to find from the Intrinsic List is its type. We do not process arguments, and Pass 2 does not know about it at all; to Pass 2, it's just another ordinary function. Therefore, once a function has been found to be basic external, that permissibility rules out basic external from then on. Since we now know the type, which was the only important thing, Pass 1 can also treat it as just another ordinary function.

Note that if the ADP (Automatic Double Precision) Trigger is set, and the function's inherent type would otherwise be Real or Complex, we have to change it to Double Precision or Double Complex, respectively. Pass 1 has the easy part of that job. Pass 2 has to actually call different system routines based on this information. (For example, a reference to SIN normally calls 9SIN, but under ADP it has to call 9DSIN.)

Intrinsic functions are handled by the routine Process If Intrinsic, called by Function Scan, which is part of Expression Scan. (See Section 3.10.1.3.4.3.)

3.6.2 Intrinsic Function Generators Pass 2)

Each of the 120 intrinsic functions has a code generator for it in Pass 2. These code generators are part of the general Arithmetic Expression generator, and they use various parts of the generator to do their work. It is approximately as if each intrinsic function were an operator (like "+" or ").

For an external function, the compiler evaluates the operand expressions from left to right in a staid, orderly way. For the intrinsics, left-to-right evaluation is not guaranteed, and the compiler makes some effort to generate good object code by considering the funtion in the light of its operands and of the expression in which it is embedded.

For certain functions the compiler is willing to evaluate the operands with the wrong sign. It knows, for example, that ABS(-X) equals ABS(X), so it will not generate a superfluous negation. Although the above example looks silly (who would write ABS(-X) anyway?), it is not so silly to save a negation in ABS(X-Y*Z). For other functions, the compiler can save negations by another identity, exemplified by SIN(-X) equals -SIN(X). Here the negation is not necessarily eliminated, but it is "postponed" to theouter expression, where it may or many not cancel with some other minus sign. Functions of which f(-x) - f(x) are called Even functions; those of which f(-x) = -f(x) are called Odd functions. Functions of which neither identity is true are not called anything in particular, but many of these, such as SQRT, are generated through a routine called LIBFGEN (Library Function Gen). Functions requiring the library, such as SIN, are generated through ODDFGEN; those such as COS, through EVENFGEN.

Functions such as MAX, with multiple arguments, inspect their arguments to decide which ones to do first in order to save generated object code. This is done in much the same way as for the "+" and "*" operators.

In Pass 2, many intrinsic functions are handled by the same routine. For example, the SQRT, DSQRT, CSQRT, and CDSQRT functions all branch to the same place, giving the impression that these functions are all the same. This is not quite true. When we arrive at a function generator, we have some information about the function on the Work List. We have the type (mode) of the function and that of its argument. These modes will effect the code being generated.

It not only saves code to combine some of these functions, but it is actually necessary in many cases to make the ADP (Automatic Double Precision) option work. If this option is specified, the SQRT function must behave like the DSQRT function. This is managed not through Pass 1's changing the number of the function in the Symbol Table to that of DSQRT, but by changing the result mode in the Function Call Plex. This way Pass 2, even though it receives the number of the SQRT function, sees that the result expected is of Double Precision type and calls SDSQRT.

We arrive at each intrinsic function generator with a pointer in WO to a Sprog Arg Group Plex. Each generator is expected to discard WO and to leave the function result in AC. It may change the bottom of the SIGNLIST if it has anything to report.

3.7 Constants

There are ten different kinds of constants recognized in the compiler:

1. Integer. E.g., 347
2. Logical. E.g., .TRUE.
3. Hexadecimal. E.g., 8Z6789ABCD
4. Left-justified Hollerith. E.g., 2HAB
5. Right-justified Hollerith. E.g., 2RXY
6. Real. E.g., 3.1417
7. Double precision. E.g., 3.1415926535700
8. Complex. E.g. (0.0,1.0)
9. Double complex. E.g., (3.40-6,9.90-7)
10. String. E.g., 'ALPHANUMERIC STRING' or 10HSAME THING

They are all scanned by Constant Scan (CONSCAN) in Pass 1. They are stored on six different lists (with some doubling up, but not enough to obsure which is which). In Pass 3, the constants that have actually been used are generated as literals following the program. Also in Pass 3, if an object listing is requested, the symbolic representations of the constants are reconstructed and printed on the listing. These four processes are discussed in the following subsections.

3.7.1 Storage

The various kinds of constants are stored on different lists, not only because they are of different sizes, but also to keep them differentiated from one another so they can be listed properly. Sometimes two different kinds of constants are kept on the same list, in which case something else (usually the mode of the list pointer to the constant) determines which kind it is. In general, each constant list entry has two parts: the value of the constant; and a trigger indicating whether the constant has actually been used. If the constant is used, the trigger location is later used to hold the location of the constant in the generated literals.

The following indicates which list each type of constant is stored on. Further information on the inidvidual lists can be found in Section 1.5.

Integer and logical constants are kept on the ICONSLST (Integer Constant List). A single entry can represent both kinds of constant (e.g. -1 and .TRUE.). The constant type is determined by the mode of the ICONSLST pointer, which is either integer or logical.

Hexadecimal constants are kept on the XCONSLST, which is exactly like the ICONSLST, including the fact that it has integer mode. The only difference is that a pointer to the XCONSLST tells Pass 3 to list the constant in hexadecimal rather than integer.

Both the H and R forms of Hollerith constants are kept on the HCONSLST, which is also exactly like the ICONSLST, including integer mode. Pass 3 determines whether to list such constants in H form or R form by examining the value of the constant. If its first byte (character) is a binary zero, then it is sensible to list it as an R form; ohterwise it is not.

Real and double precision constants are both kept on the RDCONLST. They are both kept in double precision form, because even a constant that looks like single precision may turn out to have to be double precision. (A double precision element later on in an expression forces everything into double precision.) Pass 2 may decide to use a double precision version of a constant that was registered as single precision. A constant will be registered as double precision only if it has a D exponent or if ADP mode is in effect (never by having more than a certain number of digits). This is reflected by the mode of the pointer, which will be either real or double. Each item on the RDCONLST has <u>two</u> Constant Used Triggers: one for "Used as Real", the other for "Used as Double".

Complex and double complex (komplex) constants are similarly kept on the same list, the CKCONLST. They are both kept in double complex form, and the pointer mode is either complex or komplex. CKCONLST items have two Constant Used Triggers each: "Used as Complex", and "Used as Komplex".

String constants are kept on the SCONSLST. It is the only one with a variable length entry size, since long alphanumeric strings may be of any length. Otherwise it is quite similar, having a "used" trigger and a value. The value has two parts, the character count and the characters themselves.

There is one other list that might be thought of as a constant list, the DHEXLIST This list is not used for actual constants that will ever be generated into literals, however. It is used only during the DATA statement, to hold the hexadecimal constants that may be up to four words (32 digits) long. The DHEXLIST is never used after Pass 1.

3.7.2 Scanning

Generally, the assembly CONSCAN (Constant Scan) is used to scan all ten kinds of contants. It is usually called from within an expression, where several different kinds of constants may be permissible. It also has a special entry point (RSINTSCN) for scanning only integer constants. This is used by PAUSE, STOP, and DIMENSION, which want to allow integer values, but also want to have them registered as constants (i.e., on the ICONSLST). There are other places that scan for integers, but then proceed to do something else with the values obtained, rather than make constants out of them (for example, statement numbers or FORMAT specifications). Thus we do not really consider these values as constants.

For the most part, each type of constant is written differently in the source program and is thus distinguishable by scanning. However, real constants may turn out to be double precision (e.g. from the ADP option); similarly, complex constants may turn out to be double complex.

Note that a Hollerith constant and a string constant may both have the form nH------. However, this form of long string constant can only appear in a very few contexts, where the ambiguity is not important. That is, it does not matter whether we consider 4HABCD a Hollerith or a string constant in those places where both are allowed. In fact, we consider it a string constant. But in all other places, it will come out as a Hollerith constant.

Scanning of floating point constants is done in precision greater than double precision, so that the final value will be accurate at least to double precision. The following were guidelines in the design of the floating point conversion procedures:

1. It should produce exactly the same binary values as the run-time conversion package (in 9IEDIT), when presented with the same symbolic representations.

2. The binary values should be accurate to the nearest bit; i.e., as accurate as the machine will permit. I suspect that this rule may not be universally true: that for some conversions the accuracy is <u>within a bit</u>, rather than being rounded to the exactly proper bit.

3. The accuracy will not be adversely affected by using a lot of insignificant zeros. On some compilers, a decimal point followed by fifty zeros followed by a one does not result in as accurate a value as the representation 1E-51. Probably on some compilers it results in a <u>more</u> accurate value. Neither of these conditions is desirable.

After a constant is scanned, it is registered on the appropriate list. This means that the list is searched to see if the same value is already on it. If so, processing continues using a pointer to the old value. This is how the constant .TRUE. will find that it has the same value as -1, thus resulting in a pointer to the same location on the ICONSLST, but with a different mode field in the pointer.

3.7.3 Generation

After Pass 2 has finished generating the object code, it is known which constants have actually been used, and in what forms. (A constant on the RDCONLST may have been used as real, double, both, or neither.) It is then possible to generate the literals that will appear in the object program. This is done by GENLIT (Generate Literals) at the beginning of Pass 3, and is described in detail in Section 2.27. The values are placed on the LITERLST, using an overlapped search to try to double up on words that are the same (e.g., integer and floating point zero, or a one-word Hollerith constant and part of a string constant). The constants are not removed from the various constant lists, because Pass 3, when outputing and listing, will be given pointers to the constants on these lists. However, the "constant used" word in each constant entry is no longer needed, so that space is used for the address of the generated literal. Thus from a constant list pointer, Pass 3 can determine and output the actual program location of the constant.

3.7.4 Listing

When Pass 3 is printing an object listing, it needs to be able to show the symbolic representation of the constants that have been used. It reconstructs this from the values on the various constant lists. It odes not use the LITERLST value; nor does it have access to the original source representation. Each type of constant is listed in its appropriate source language format, which may be decimal, hexadecimal, alphanumeric, logical, or complex. This process is described in Section 2.29.3.3.2.

3.8 Allocation

Aside from the scanning of COMMON,EQUIVALENCE and GLOBAL statements (see Sections 3.29, 3.48, and 3.53), all the work of Storage Allocation is done in an assembly called ALLOCATE. ALLOCATE is called just before PASS2, and it is ALLOCATE's job to figure out where all the variables are to be stored.

Mainly because of the EQUIVALENCE statement, this is a rather horrendous task. ALLOCATE is some 1962 lines long and occupies 1074 words. The main reason for such length is the detailed error messages that ALLOCATE is expected to produce.

For example, because of hardware restrictions, we are not allowed to allocate double precision, complex, or double complex variables in odd locations. Since the EQUIVALENCE statement can imply such an allocation, either directly or indirectly, we may produce such a message as

```
WARNING:  EQUIVALENCE OF 'X' TO 'Y' ON LINE 23 IGNORED.
          'A' AND 'B' MUST BOTH LIE ON DOUBLEWORD BOUNDARIES.
```

There are seven different reasons why an equivalence of two variables may be ignored, and the same equivalence may be ignored for more than one reason, in which case all the reasons are printed. (If we did not print multiple reasons, the user could conceivably correct his program only to have the same equivalence ignored for a different reason.)

From our observations, EQUIVALENCE is used fairly rarely in FORTRAN programs, and it is probable that our error diagnosis represents an overkill of the problem.

ALLOCATE begins with a sweep through the Symbol Table to zero the last two words of eaach item (YALOCPTR and YALODISP). These words will carry the information about where the variable will be stored.

After the symbol table sweep, we call a routine called Allocate Blank COMMON (ALBLNCOM), then one called Allocate Labeled COMMON (ALLABCOM), then Process EQUIVALENCE (PROCEQIV). (Most of the really complicated processing happens in PROCEQIV.) Thus we have handled, in order, those COMMON statements dealing with blank COMMON, those COMMON statements dealing with labeled COMMON, and all EQUIVALENCE statements. Any of these may produce diagnostics.

Then we call Allocate Local Variables (ALLOCALV) and Allocate Dummies (ALDUMMYS). Allocation of the local variables needs the information supplied by EQUIVALENCE statements, and allocation of dummies needs the information supplied by ALLOCALV in that the dummies are allocated following the local variables. Then comes a routine called Check COMMON Parity (CKCOMPTY). This checks to see if EQUIVALENCE has caused any COMMON blocks to need to start in odd locations. If a COMMON block needs this, a diagnostic is printed and we insert an extra word at the beginning of the block and slide all the variables up by one word.

Then we call two routines named Finish Blank COMMON (FINBLCOM) and Finish Labeled COMMON (FINLBCOM), which mainly check for overflow of the various COMMON blocks and complain if it occurs. Then we destroy the EQUIVALENCE data by PAPing the EQUIVALENCE Clump List (CLUMPLST) and the EQUIVALENCE Chain List (EQVCHLST), and then we exit.

3.8.1 Initialize Symbol Table INISYMTB _ ALLOCATE)

Basically, INISYMTB is a merry little sweep through the Symbol Table zeroing a couple of rungs. The only ugliness comes if any of the symbols happen to be GLOBAL, and this will be discussed later.

We step through the SYMTABLE one item at a time and zero the YALOCPTR (Allocation Pointer, the former Permissibility word) and YALODISP (Allocation

Displacement) rungs. The Allocation Pointer rung tells what class of stroage the variable (if it is a variable) is assigned to--such as blank COMMON, a particular COMMON block (a pointer to the BLNAMLST), local variable storage. The Allocation Displacement rung is a relative location within that storage class.

Zeroing the Allocation Pointer means that the variable is Free--that it has been assigned to no storage class. By the end of ALLOCATE, there will be no Free variables. (There may be a few Free subprogram names, which Pass 2 will give locations in Program Storage.) Before a variable becomes Allocated (where YALOCPTR points to something specific), it may become Tied--meaning that it has been EQUIVALENCEd to one or more other variables. In this case YALOCPTR is a pointer to the EQUIVALENCE Chain List (EQVCHLST); such a pointer is converted to something else before the end of ALLOCATE.

Now the GLOBLA ugliness. A GLOBAL variable is handled much as if it were a COMMON block with the same name as the variable. Thus we have a GLOBAL Name List (GLNAMLST) with a structure parallel to the Block Name List (BLNAMLST). Not all of the GLOBAL variables get their names put

on this list, however. A GLOBAL variable needs to have its name on the GLNAMLST only if it is EQUIVALENCEd to something.

There are times during ALLOCATE when we want to know if a thing is either GLOBAL or EQUIVALENCEd to GLOBAL. To check for this, we look at YALOCPTR in the Symbol Table and see if it is a pointer to the GLNAMLST. For this reason, we want all GLOBAL symbols to have such a pointer, even if the pointer is not valid. (If a valid pointer is needed, it will be created later.) So on the initial sweep through the Symbol Table, if the symbol is GLOBAL (tested with SOF), we store a pointer to the GLNAMLST in YALOCPTR instead of zero.

3.8.2 Allocate Blank COMMON ALBLNCOM _ ALLOCATE)

Allocate Blank COMMON is quite similar to Allocate Labeled COMMON (ALLABCOM, 3.8.3), the main difference being that labeled COMMON has to worry about several different COMMON blocks while blank COMMON has only one. It might seem that Allocate Blank COMMON should be shorter than Allocate Labeled COMMON, but in fact Allocate Labeled COMMON is two POPs shorter than Allocate Blank COMMON (44 POPs to 42). This is partially because Allocate Labeled COMMON is coded a little tighter than Allocate Blank COMMON, but mainly because Allocate Blank COMMON has to do some initializing and finalizing that the other routine does not have to do. For labeled COMMON, such information as size, parity, parity culprit (a parity culprit is a symbol symbol table pointer to a variable that has established a parity preference for a COMMON block or an EQUIVALENCE clump; it is used in error messages for parity violations) and block overflow trigger are kept in various rungs on the Block

Name List (BLNAMLST); for blank COMMON, the same information is kept in temps during allocation, then entered into a siz-word item on the Blank COMMON List (BLCOMLST) at the end. These things must be initialized for blank COMMON, but for labeled COMMON Pass 1 has already initialized it on the BLNAMLST.

ALBLNCOM zeroes or MEFs the temps just mentioned, then makes a destructive pass through the BLCOMLST, taking off variable names from the top and allocating them one by one. These are all allocated at the size they were DIMENSIONed and simply follow each other in increasing memory locations. The only deviation from this plan is when a parity violation occurs. If there is only one parity violation, the problem can be fixed by sliding all of blank COMMON en masse upward one word. This is handled later in Check COMMON Parity (CKCOMPTY, 3.8.7); it is handled later rather than now because a COMMON block may acquire an odd parity preference through either COMMON statements or EQUIVALENCE statements. Complaint is through ERROR11.

If two parity violations occur, i.e., two variables requiring even locations must lie an odd distance from each other, then ALBLNCOM gets annoyed. It complains about this with ERROR7:

WARNING: THE { DOUBLE PRECISION / COMPLEX / DOUBLE COMPLEX } VARIABLE 'D' IN { BLANK COMMON / COMMON BLOCK /BBB/ } ...
MUST LIE ON A DOUBLEWORD BOUNDARY.
IT AND EVERYTHING BEYOND IT WILL BE SHIFTED UP BY ONE WORD.

then adds one to BLCOMSZ before allocating the variable. In a particularly perverse COMMON statement, this can happen every second variable.

At the end of processing of the BLCOMLST, the list is empty and we now create on it a six-word item consisting of the following things

0. Block Name Rung (all blanks) YBLNAME
1. (2nd half of name) -
2. Block Size Rung YBLSIZE
3. Block Parity Rung YBLPRTY
4. Block Parity Culprit Rung YBLPRTYC
5. Block Overflow Trigger Rung YBLOVFL

This item is parallel to items on the Block Name List (BLNAMLST) and the GLOBAL Name List (GLNAMLST) and is for the convenience of Process EQUIVALENCE (PROCEQIV, 3.8.4 and subsections).

3.8.3 Allocate Labeled COMMON ALLABCOM _ ALLOCATE)

ALLABCOM is very similar to Allocate Blank COMMON. The main difference is that the input data is on the Labeled COMMON List (LBCOMLST) instead of the Blank COMMON List, and the data is more complicated: instead of a list of just variables we have a list with block names interspersed with the variables. The first item on the LBCOMLST must be a block name.

ALLABCOM does no initializing or finalizing and just makes its way through the Labeled COMMON List until it is done. It makes the same decisions about parity culprits and can cause the error messages already described to be printed. It keeps going on one block name until a new block name is encountered.

3.8.4 Process EQUIVALENCE PROCEQIV _ ALLOCATE)

In spite of its rather grandiose sounding name, Process EQUIVALENCE does not do much. It is mainly a front man for lower level routines that do all the work. First we FLIP the EQUIVLST from which we will get our data. This is done so we can get our data with MOFs instead of TOTs. The only reason that MOFs are preferable is that occasionally (down in Next Equiv Item, 3.8.4.2) we want to MOF something, look at it, then put it back with MON. There is no inverse TOT POP with which we could put it back if we did not the list.

Then we start MOFing items from the EQUIVLST and SOT for EQSETBIT. This is an ID field meaning "start an EQUIVALENCE group and here is the line number for it". If we do not find an EQSETBIT, we abort. If we do, we mask away the ID field and store this in EQUIVALENCE Statement Line Number (EQLINENO) for possible later inclusion in error messages. Then we JRS to Process EQUIVALENCE Group (PEQGROUP, 3.8.4.1) which does not do very much either.

3.8.4.1 Process EQUIVALENCE Group PEQGROUP _ ALLOCATE _

This routine is responsible for breaking down an EQUIVALENCE group into a series of EQUIVALENCE pairs. For purposes of diagnosis, an EQUIVALENCE group of more than two elements is considered to imply the EQUIVALENCE of every pair of elements in the group. For example, the statement

```
EQUIVALENCE     (A, B, C, D, E)
```

is taken to mean

EQUIVALENCE (A,B), (A,C), (A,D), (A,E), (B,C),

(B,D), (B,E), (C,D), (C,E), (D,E)

If, by great ingenuity, the programmer comes up with A, B, C, D, and E such that no two can be EQUIVALENCEd, the compiler prints a diagnostic concerning each of the ten pairs above, telling why those two could not be EQUIVALENCEd. If any pairs can be EQUIVALENCEd, they are EQUIVALENCEd. Hence the user is guaranteed at least one message for every EQUIVALENCE pair not performed.

In doing the statement

EQUIVALENCE (A, B, C, D, E)

the compiler attempts the pairs

(A,B), (A,C), (A,D), (A,E) .

If all of these succeed, we are done; to attempt the other six pairs would be redundant. If any of the above do not work, however, the compiler goes on and attempts

(B,C), (B,D), (B,E), (C,D), (C,E), (D,E)

in that order, diagnosing any that do not work.

3.8.4.2 Next Equiv Item NXEQITEM _ ALLOCATE __

NXEQITEM delivers the next item in an EQUIVALENCE group, where an item consists of a variable name and all of its subscripts. NXEQITEM combines all the subscripts into a single number called the Script Displacement, which is a displacement in words from the first word of the variable. Thus if V is a real scalar, V(1) has a displacement of zero, V(2) a displacement of one, etc. If D is a double precision scalar, D(1) has a displacement of zero and D(2) a displacement of 2.

NXEQITEM checks whether the number of subscripts agrees with the number of dimensions (of an array). If they disagree, then the number of subscripts had better be 1. If not, we produce a diagnostic of the form:

WARNING: APPEARANCE OF 'ABD' IN EQUIVALENCE ON LINE 25 IGNORED.

HAS 2 SUBSCRIPTS; SHOULD HAVE 3 OR 1 OR NONE.

Naturally, NXEQITEM will accept a variable without subscripts; in this case it gives a script displacement of zero.

Next Equiv Item delivers three words on the bottom of the WORKLIST. Upon return, W0 = Script Displacement In Words, W1 = the size of the variable in words, and W2 = a SYMTABLE pointer to the variable.

When NXEQITEM discovers an item incorrectly subscripted, it pretends the item was not there. Thus it goes back to the top of itself and tries to get another item. If there are no items to get, it returns with an answer FALSE. If all the itmes in an EQUIVALENCE group are bad, that group will appear to have no items at all. The calling routine (Process EQUIVALENCE Group, PEQGROUP) does not mind.

When single subscripts are used on variables other than single-dimensioned arrays, they have a special meaning. V(1) always means the first element of the variable, V(2) the second element, etc. If V is a single-dimensioned array, V(1) is the element with subscript value 1. Thus if V is dimensioned

```
DIMENSION V(0:5)
```

then V(1) is not the first element. But if we dimension V

```
DIMENSION V(0:5,2)
```

then V(1) is the first element. This convention (which is true only in EQUIVALENCE statements) is a carry-over from FORTRAN II, which did not allow multiple subscripts in EQUIVALENCE statements but only the single-number form, indicating a displacement from the first element. FORTRAN IV allows either form, and because our FORTRAN IV allows subscript lower bounds other than 1, we have an inconsistency between the two forms. NXEQITEM takes care of the inconsistency.

3.8.4.3 Left and Right

The two variables to be EQUIVALENCEd by Process Equiv Pair (PEQPAIR) are called the Left variable and the Right variable. These names correspond to the appearance of the variables in the EQUIVALENCE statement. It takes three words to hold the data about a variable: the ID Pointer, the Size, and the Script Displacement. These three words are delivered by Next Equiv Item (NXEQITEM).

Naturally we have a Left ID Pointer and a Right ID Pointer, a Left Size and a Right Size, and a Left Script Displacment and a Right Script Displacement. Also, in case either Left or Right happens to be previously EQUIVALENCEd, we have a Left Clump Pointer and Right Clump Pointer, each pointing to an item on the CLUMPLST telling about the EQUIVALENCE clump.

In EQUIVALENCE The Pair (EQVPAIR, 3.8.4.5) the goal is to get the Left variable to be the more firmly anchored of the two, where increasing order of

"anchoredness" is: Free, Tied, Allocated. If it happens that the Right variable is more firmly anchored than the Left, we will swap the two by calling the subroutine Swap Left And Right (SWAPLR, 3.8.4.5.1) which not only swaps the four temps mentioned above but also reverses the Left-And-Right-Swapped Trigger (SWAPTRIG). It reverses this trigger instead of setting it, because the same routine is used to swap them back. The SWAPTRIG is interrogated only by the error message printing routines, because these want to print the variables in the order they appeared in the source statement.

The fact that we can swap the Left and Right variables reduces considerably the number of cases that have to be handled.

3.8.4.4 Process Equiv Pair PEQPAIR _ ALLOCATE _

This routine is the heart of the whole EQUIVALENCE operation. It is quite easy to say what Process Equiv Pair does, but the details of how it does it run on for many pages of code. PEQPAIR's job is: take the two variables called Left and Right and EQUIVALENCE them if possible.

If the EQUIVALENCE is not possible, then print the appropriate diagnostic message(s) and exit with the EQUIVALENCE Pair Error Trigger (EQPERTRG) and the EQUIVALENCE Group Error Trigger (EQGERTRG) set. The EQUIVALENCE Pair Error Trigger (EQPERTRG) is initialized FALSE by Process Equiv Pair, but the EQUIVALENCE Group Error Trigger (EQGERTRG) is initialized outside this routine and accumulates an answer from serveral EQUIVALENCE pairs.

Most of the actual work of Process Equiv Pair is done in a subroutine called EQUIVALENCE The Pair (EQVPAIR, 3.8.4.5); the outer initializes two triggers (the EQUIVALENCE Pair Error Trigger (EQPERTRG) and the Left-And-Right-Swapped Trigger (SWAPTRIG) FALSE, then calls the inner routine EQVPAIR. After returning from EQVPAIR, it tests the Swap Trigger (SWAPTRG) and, if TRUE (indicating that Left and Right have been swapped), calls SWAPLR (Swap Left And Right) to swap them back again. This is necessary because Process EQUIVALENCE Group (PEQGROUP, 3.8.4.1) expects these to be unswapped.

The last thing PEQPAIR does before exiting is to try to save some room on the Clump List (CLUMPLST). This list holds information about variables that are EQUIVALENCEd to each other, and sometimes, when a clump is allocated into COMMON, the need for the information on the CLUMPLST vanishes. If this clump happens to be bottom-most on the CLUMPLST, we can get

* The EQUIVALENCE Pair Error Trigger (EQPERTRG) and the Left-And-Right-Swapped rid of that clump. Clumps are eight words long and by convention, when the clump dies, the first rung is set to zero. So we look eight words up from the bottom and see if that word is zero and if so we reduce CLUMPBOT by eight. This is all done through a parameter called YYCLUMP (size of a CLUMPLST item) that is defined in LISTDEFS to be eight.

This feeble attempt to save working storage is not very useful. The CLUMPLST seldom gets very large anyway, and the code required to check for dead clumps is more than the size of a clump. In any case, the whole Clump List gets ZAPed at the end of ALLOCATE.

3.8.4.4.1 Data Structure for EQUIVALENCE Clumps

An EQUIVALENCE clump is a set of variables that have been EQUIVALENCEd to each other. This information is supplemented with information about where the variables are stored relative to each other. Then there is information about the clump as a whole, such as its size, and whether or not it has a parity preference (preference to begin at an odd or even location) and if so, what variable caused it to have that preference.

The information about a clump is stored on three lists: the Clump List (CLUMPLST), the Symbol Table (SYMTABLE) and the EQUIVALENCE Chain List (EQVCHLST). Detailed information about the structure of these lists is given in Section 1.5.

Of these three lists, the EQUIVALENCE Chain List is almost redundant. The only reason that the EQVCHLST exists at all is that ALLOCATE needed another word in the SYMTABLE, and it did not seem worthwhile to add one for such a limited purpose. If there were an extra word on the SYMTABLE, then Rung 4 (YALOCPTR) would point to the next SYMTABLE item in the chain, and Rung 6 (the new rung which we would call YCLUMPTR) would point to the CLUMPLST. Rung 4 would be serving the function of YCHAINPT.

The problem that the EQVCHLST solves is that, for every Symbol Table entry, we need a pointer to tell what clump it is in, if any, as well as a pointer to the next item in the chain. Three lists are involved in the representation of an EQUIVALENCE clump, and the following diagram may help you understand how three tables are intertwined. The data shown is for the EQUIVALENCE of three variables. This involves three items on the Symbol Table, three items on the EQUIVALENCE Chain List, and one item on the EQUIVALENCE Clump List. The statement represented is

```
EQUIVALENCE (P, Q(3), R(-4))
```

where P, Q, and R are scalars of one word apiece. If they were arrays whose lowest subscript bound was 1, the only possible change in this data would be in YHIELEM; lower bounds other than 1 might alter YLOWELEM, but the point is that this data structure would be true for scalars or arrays.

3.8.4.5 EQUIVALENCE The Pair EQVPAIR _ ALLOCATE

We have already explained the essence of what EQVPAIR does; here the details are given. ALLOCATE has eleven error messages, of which seven are detected in EQVPAIR.

There are nine essential cases for EQVPAIR to consider: Left and Right may each be Free, Tied, or Allocated. EQVPAIR reduces these to two basic cases--one where Left is Allocated and Right is Tied, and the other where Left is Tied and Right is Tied. The Free cases are eliminated by building any free variables into one-item clumps (see 3.8.4.5.2 and 3.8.4.5.3). Some of the cases are lumped together by exchanging Left and Right (see SWAPLR, 3.8.4.5.1).

The following is a flowchart of EQVPAIR. A few of the labels used in the code appear in the flowchart (HRCLUMP, 3Z7, 3Z10 and 3Z11), and many of the boxes agree with the comments in the listing.

There is a reason that the remarks in most of the flowchart boxes start with lower-case letters: the only things that are capitalized are proper names such as Right and Left--names that are used in the code. If we started sentences with initial caps, it would be unclear which names were proper and which were not.
Error messages 1 through 6 all start with the line

WARNING: EQUIVALENCE OF 'XXX' TO 'YYY' ON LINE 25 IGNORED.

and then continue with a second line explaining why. All of these messages set the EQPERTRG and EQGERTRG, thus causing the EQUIVALENCE to be ignored. The second lines of the first six error messages are as follows.

ERROR 1 CONTRADICTS PREVIOUS EQUIVALENCE.

ERROR 2 CONTRADICTS PREVIOUS ALLOCATION IN

ERROR 3 'XXX' IS

ERROR 4 EXTENDS

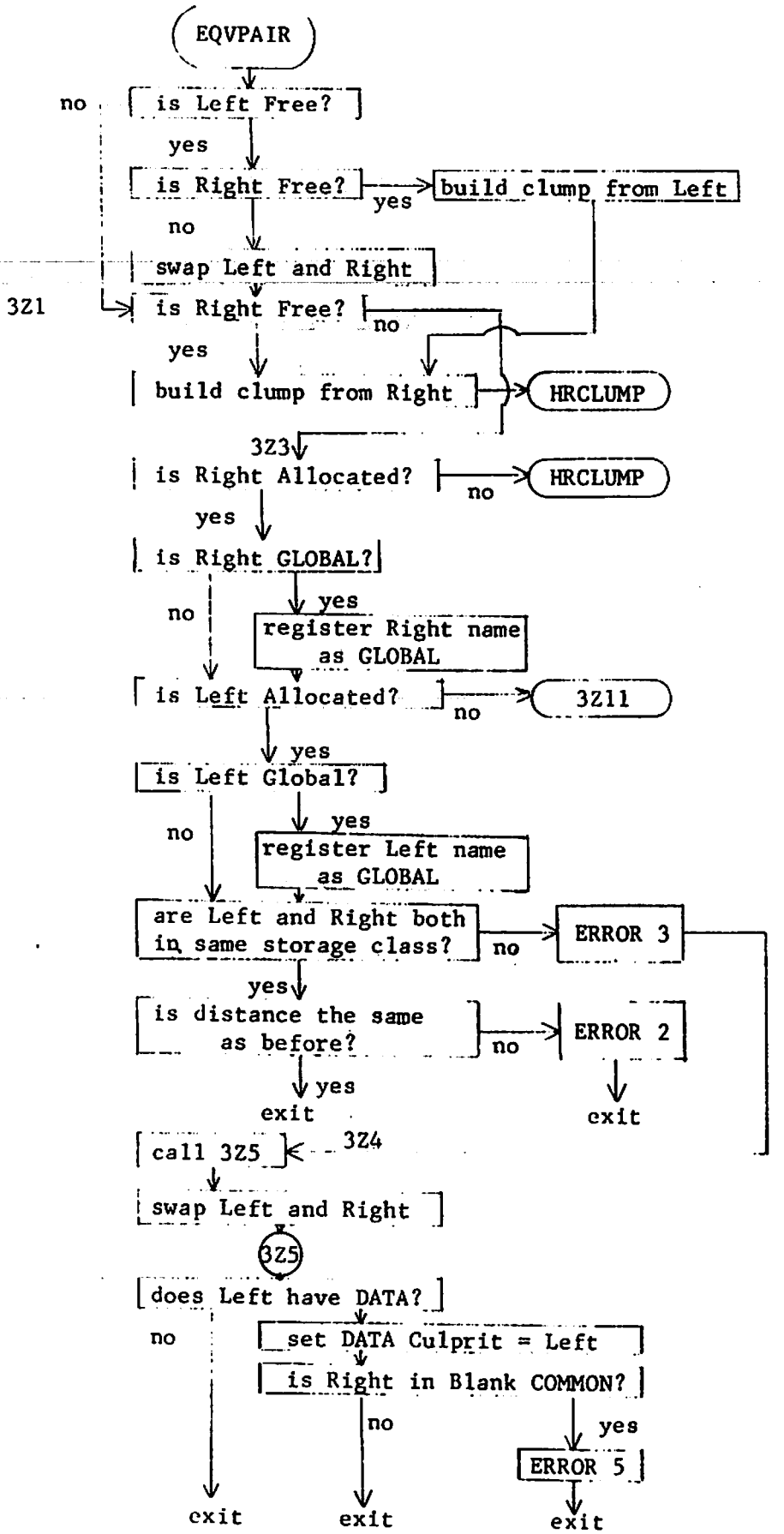
ERROR 5 'VVV' IS IN A DATA STATEMENT AND CANNOT BE IN BLANK COMMON.

ERROR 6 'VVV' AND 'WWW' MUST BOTH LIE ON DOUBLEWORD BOUNDARIES.

Error message 10 is similar to 1 thru 6 but differs in two important ways. The most important is that it does not set the EQPERTRG and EQGERTRG triggers, and thus the occurance of this error does not prevent EQUIVALENCE. Thus the first line of the message does not say "IGNORED". The message reads

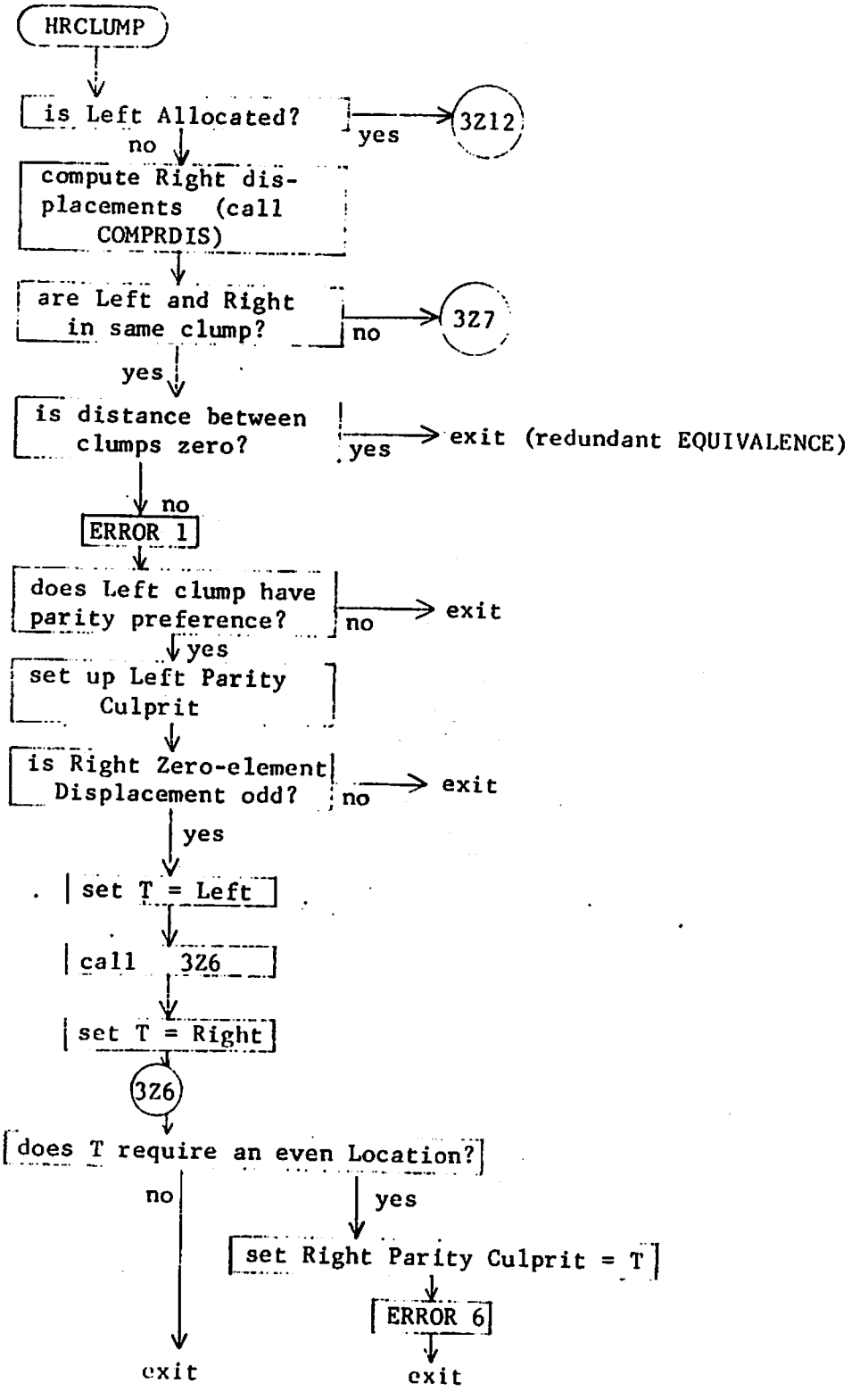
```
WARNING:  EQUIVALENCE OF 'XXX' TO 'YYY' ON LINE 25
          CAUSES EQUIVALENCE SET TO EXCEED MAXIMUM MEMORY SIZE.
```

Later, when this equivalence set gets allocated, another message will appear saying that blank COMMON or local variable storage or whatever has exceeded maximum memory size (ERROR 9).

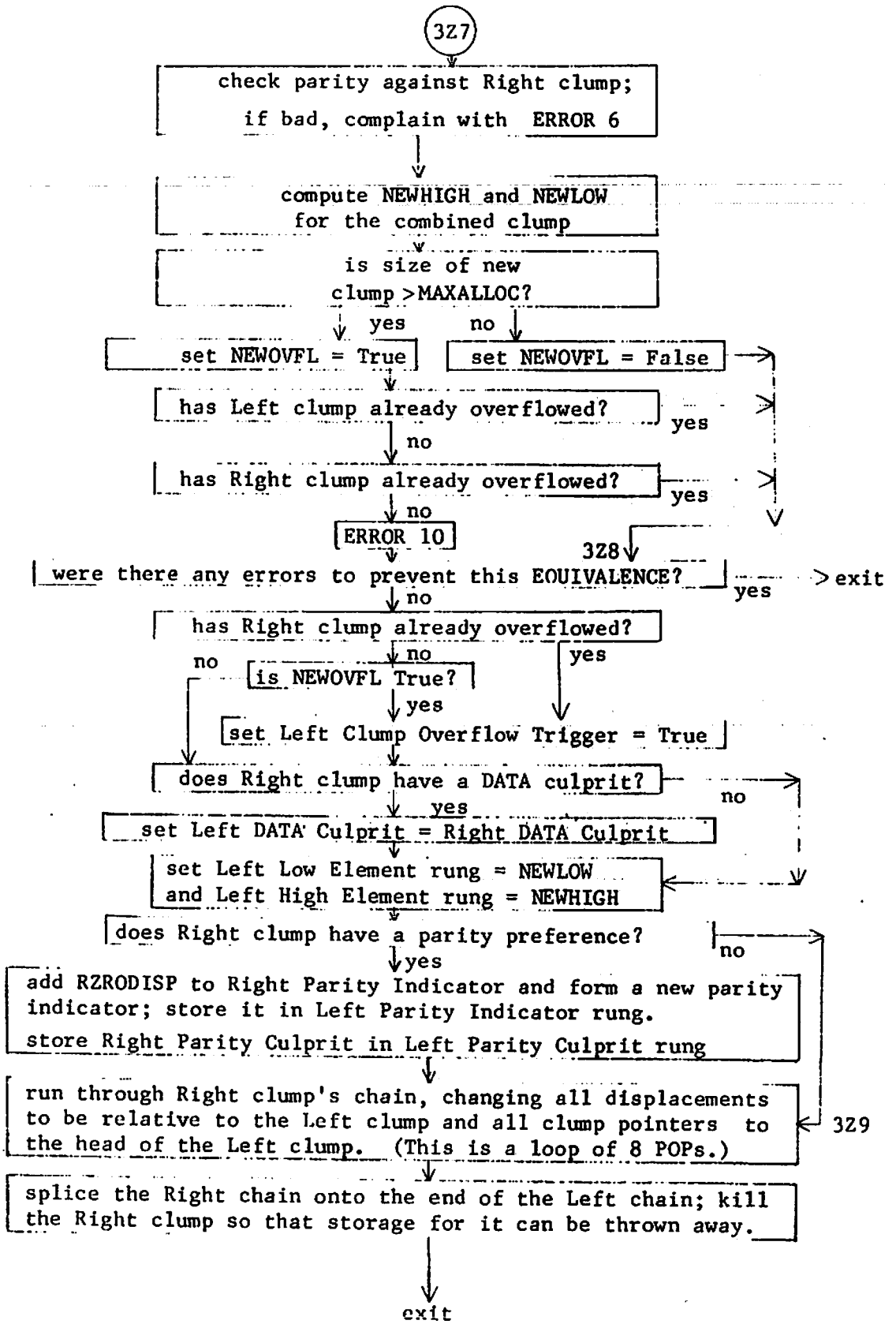


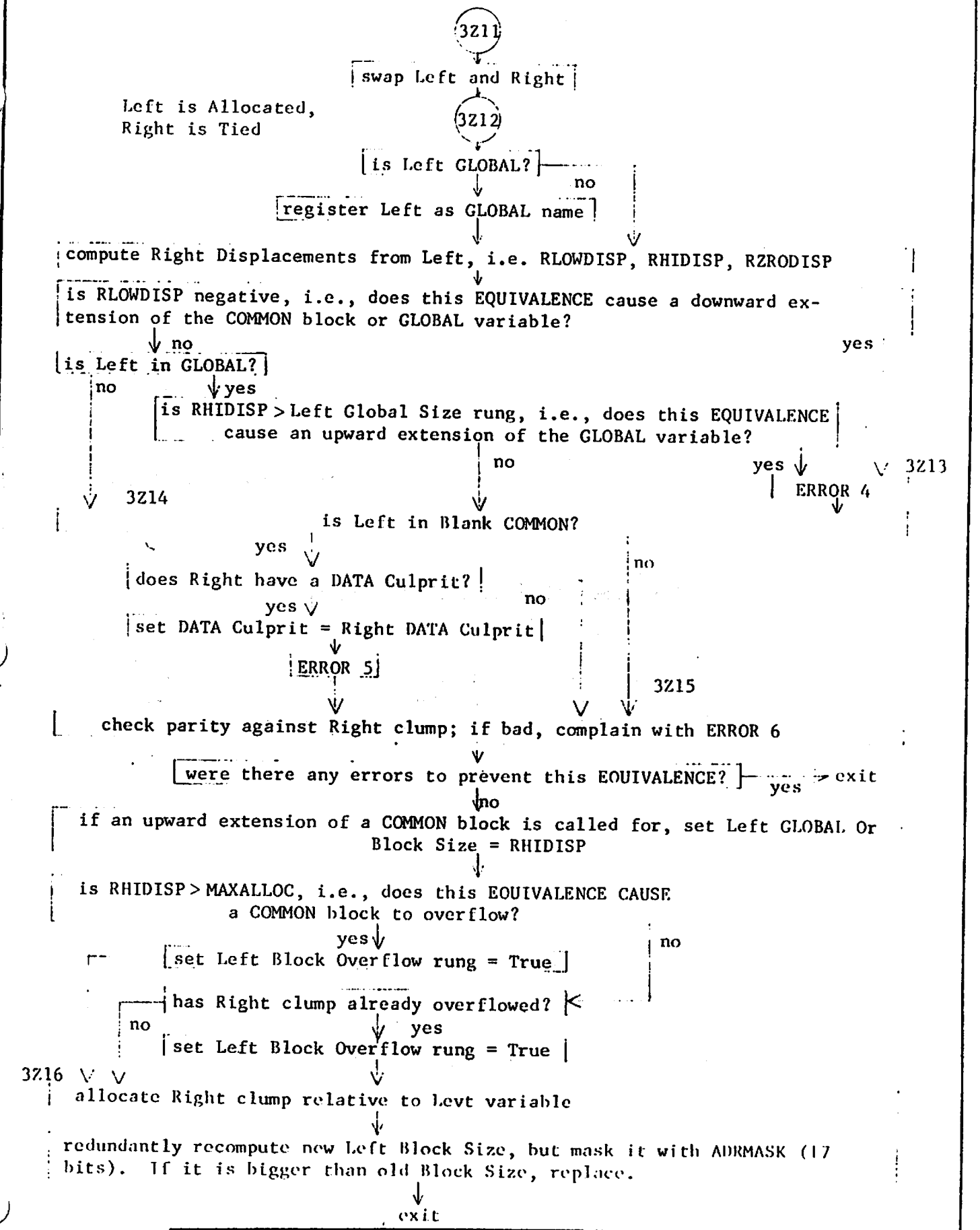
TITLE SIGMA 5-9 XEROX EXTENDED FORTRAN IV TECHNICAL MANUAL	XEROX TEXT/XGP
	DRAWING NUMBER: 703270 A
	PAGE 445a

Handle Right Clump (HRCLUMP)



Left and Right are both Clumps




3Z11
swap Left and Right
Left is Allocated,
Right is Tied
3Z12
is Left GLOBAL?
no
register Left as GLOBAL name
compute Right Displacements from Left, i.e. RLOWDISP, RHIDISP, RZRODISP
is RLOWDISP negative, i.e., does this EQUIVALENCE cause a downward extension of the COMMON block or GLOBAL variable?
yes
no
is Left in GLOBAL?
no
yes
is RHIDISP > Left Global Size rung, i.e., does this EQUIVALENCE cause an upward extension of the GLOBAL variable?
no
yes
3Z13
ERROR 4
3Z14
is Left in Blank COMMON?
yes
no
does Right have a DATA Culprit?
yes
no
set DATA Culprit = Right DATA Culprit
ERROR 5
3Z15
check parity against Right clump; if bad, complain with ERROR 6
were there any errors to prevent this EQUIVALENCE?
yes
exit
no
if an upward extension of a COMMON block is called for, set Left GLOBAL Or Block Size = RHIDISP
is RHIDISP > MAXALLOC, i.e., does this EQUIVALENCE CAUSE a COMMON block to overflow?
yes
no
set Left Block Overflow rung = True
has Right clump already overflowed?
no
yes
set Left Block Overflow rung = True
3Z16
allocate Right clump relative to Levt variable
redundantly recompute new Left Block Size, but mask it with ADRMASK (17 bits). If it is bigger than old Block Size, replace.
exit

3.8.4.5.1 Swap Left And Right SWAPLR _ ALLOCATE)

This routine exchanges the content of four pairs of temps in order to simplify some of the cases of EQUIVALENCE pairs. We swap

Left ID Pointer	and	Right ID Pointer
Left Size	and	Right Size
Left Script Displacement	and	Right Script Displacement
Left Clump Pointer	and	Right Clump Pointer

We also exclusive-OR the constant TRUE into the Left-And-Right-Swapped Trigger (SWAPTRIG) so we can keep track of how many times the variables have been swapped. Thus Swapping twice is equivalent to not swapping at all. See 3.8.4.3.

Since EQUIVALENCE is a commutative operation, swapping the two variables makes no difference; but the error printing routines want to print the variables in the order they were written, so they interrogate the SWAPTRIG.

3.8.4.5.2 Build Clump From Left BILCLMPL _ ALLOCATE)

This subroutine simplifies some of the work done by EQUIVALENCE The Pair. The Left and Right variables may each be Free, Tied, or Allocated. Build Clump From Left allows us to forget the fact that a varaible may be Free. If a variable is Free, we build it into a single-item clump and thereafter treat it as if it were Tied. This causes extra work to be done, but it simplifies the coding.

The most complicated cases that naturally arise are Tied versus Tied and Tied versus Allocated. In Tied versus Tied we have to combine two EQUIVALENCE clumps, and in Tied versus Allocated we have to allocate a clump. We essentially reduce all the Free cases to these two cases, then use the machinery for handling these worst cases.

3.8.4.5.3 Build Clump From Right BILCLMPR _ ALLOCATE)

This routine is kind of fun, because it consists of just two POPs and a fall-in (i.e., it falls into another routine). It JRSs to Swap Left And Right, then JRSs to Build Clump From Left, then falls into Swap Left And Right. So two swaps are carried out, and the clump is built from the left.

3.8.4.5.4 Handle Right Clump HRCLUMP _ ALLOCATE

HRCLUMP is a label within EQUIVALENCE The Pair (EQVPAIR, 3.8.4.5) and not a subroutine in its own right. It is the place where the real work of EQVPAIR is done, because all the other meaningful cases of equivalencing are transformed into one where the Right variable is Tied, i.e., part of a clump. The Left variable is either already Allocated (in COMMON or someplace) or it is also Tied. As has been said before, any Free variables have been reduced to Tied by the simple device of building them into one-time clumps.

The flow of HRCLUMP is complicated and involved and consists of the last three pages (out of four) of the flowchart for EQVPAIR (3.8.4.5).

3.8.4.5.5 Compute Right Displacements COMPRDIS _ ALLOCATE)

When the Right variable is Tied and the Left variable is either Tied or Allocated, it becomes necessary to know where the Right clump is to be allocated relative to the origin of the Left thing. This means computing three displcements: the Right Low Element Displacement (RLOWDISP), Right Zero Element Displacement (RZRODISP) and Right High Element Displacement (RHIDISP). These are all temps, not rungs. RHIDISP is actually one greater than the displacement of the highest element in the clump.

In addition to computing the displacements above, COMPRDIS retrieves the Clump Pointer for the from the list it is on (in the EQVCHLST, pointed to by the Right SYMTABLE item) and stores it in a temp called Right Clump Pointer (RCLUMPTR).

3.8.4.5.6 Check Parity Against Right Clump CHEKPRTY _ ALLOCATE)

This routine is simple. It c hecks to make sure both the left and right clumps have a parity preference (if not, it exits), then adds the parity indicators together and adds in the displacement of the right clump from the left. If the result is odd, we JRS to ERROR6--the error about parity in EQUIVALENCE.

3.8.4.5.7 Allocate Equiv Clump ALOCLUMP _ ALLOCATE)

To allocate and EQUIVALENCE clump means to take all the variables in that clump and give them specific locations. These locations may be in COMMON, GLOBAL, or local storage. The clump already contains the information of where the variables are to lie relative to each other; allocating the clump merely ties these locations down to something absolute. (Absolute, that is, except for later relocation by the loader.)

There are two circumstances under which a clump may have to be allocated: (1) during EQUIVALENCE, when a variable in a clump is equivalenced to a variable that has already been allocated in COMMON or GLOBAL, and (2) during Allocate Local Variables (ALLOCALV, 3.8.5), when we run across a variable in the Symbol Table that is a member of some clump; in this case, we allocate the whole clump.

There are four parameters to ALOCLUMP. WO contains a pointer to the CLUMPLST to the clump that is to be allocated. P0 is an allocation trait (such as BLANKCOM, LABCOM, LOCALVAR) to be REGed for each variable in the clump; P1 is an allocation pointer to be stored in YALOCPNTR of each Symbol Table entry; P2 is the displacement of the lowest element in the clump (relative to the COMMON block or local storage or whatever).

The actual allocating is a matter of moving through the chain of variables that form the clump. The code for this is rather obvious, once you know the data format for a clump. This format is quite complicated and is described in Table 3.8.1. If you understand it, you should have no trouble following the listing for ALOCLUMP.

After the main loop of passing through the chain, we do some testing to see whether the clump has already overflowed (exceeded memory size). If it has, or if allocating the clump in some storage area causes that storage area to overflow, then we set the size of the storage area to the maximum allocatable size. This size is returned from ALOCLUMP in WO. The reason we do such multiple testing is that sizes may turn out to be enormous (due to keypunching errors in the source program) and to do arithmetic on several such sizes might produce an overflow that would go undetected. Thus each entity (be it clump or COMMON block) has a trigger indicating that overflow has occured, and multiple overflows will not cancel each other out.

3.8.5 Allocate Local Variables ALLOCALV _ ALLOCATE

This routine might well be called "Allocate the Leftovers." We scan through the Symbol Table looking for unallocated non-dummy scalars and arrays and allocate them. Having found one, we check to see if it is Tied; if it is Tied, we allocate the whole EQUIVALENCE clump right then and there. Then later, when we come upon (in the Symbol Table) the other variables in the clump, we will find them already allocated.

If the variable is not Tied, we check to see if it requires an even location. If so, and the location counter (LOCVARSZ) is odd, it is incremented by one, wasting a word.

3.8.6 Allocate Dummies (ALDUMMYS) (ALLOCATE)

Allocating dummies is easy, mainly because they cannot appear in COMMON or EQUIVALENCE. The temp VSIZE is already pointing one beyond the last local variable. Each dummy requires one word, so we step VSIZE by one for each dummy. This is a simple sweep throughout the Symbol Table checking for the DUMMY trait. Each time we find one, we allocate a word for it. We set the YALOCPTR rung of each symtable item to LOCVARPT, indicating that it is in local storage. And we set YALOOISP to the current VSIZE.

3.8.7 Check COMMON Parity CKCOMPTY ALLOCATE)

Check COMMON Parity consists of two calls on Check And Diagnose Odd Allocation (CHECKODD). The second call is accomplished by falling into CHECKODD. CHECKODD expects WO to contain a pointer to either the BLNAMLST or the .BLCOMLST. We TPT the BLNAMLST and, it TRUE, we JRS to CHECKODD. (If FALSE, then the BLNAMLST was empty and no pointer was created.) Then we TPT the BLCOMLST, knowing that there will be exactly one item thereon so the answer will come back TRUE. We do not check the answer and simply fall into CHECKODD.

CHECKODD examines the Block Parity Rung (YBLPRTY for each COMMON block to see if it is 3. If it is, then the block must start in an odd location, and that is now allowed. So we print a complaining message and move the whole COMMON block up by one word. This means we have to look through the symbol table for all variables allocated in this COMMON block and increment their locations by one. We also step the size rung (YBLSIZE) for each block.

CHECKODD JPNs to work on the next COMMON block, if any, and exits when through.

3.8.7.1 The Parity Problem In COMMON Blocks

The original design of ALLOCATE was based on a misconception of the way COMMON blocks (i.e., dummy sections) are handled by the loader. The notion was that anything (a COMMON block or an EQUIVALENCE clump) could have eigher an odd or or even parity preference, or no preference at all. This means that it was perfectly all right for a COMMON block to have one double precision variable that began at an odd location relative to the start of the COMMON block. All we would have to do would be to load the whole COMMON block starting at an odd location, and the variable would come out at an even location. The only thing that was forbidden was to have two doubleword variables in the same COMMON block or EQUIVALENCE clump that were required to be an odd distance from each other, for this would force one of them into an odd location.

We thought that the binary language would allow us to tell the loader to begin a DSECT at an odd location, and this is not so. (Note that we can allocate an EQUIVALENCE clump in local variable storage starting at an odd location if we want to.) That is the reason we have this routine, which notices COMMON blocks preferring odd parity and complains about them (ERROR 11) with the message

Warning: THE { DOUBLE PRECISION / COMPLEX / DOUBLE C } VARIABLE 'D' MUST LIE ON A DOUBLEWORD BOUNDARY.

{ COMMON BLOCK /B/ / BLANK COMMON } WILL BE SHIFTED UP BY ONE WORD

CKCOMPTY then goes through all the symbols in that COMMON block and adds 1 to their locations, thus increasing the size of the COMMON block by one.

This error recovery will work if the COMMON statements in all the different subprograms are the same, for then the same "correction" will be applied to all of them.

Note that this test is made after both COMMON and EQUIVALENCE have been processed, since a COMMON block may acquire a parity preference through either COMMON or EQUIVALENCE. Note that the error just described (ERROR 11) causes a whole COMMON block to be shifted and is different from ERROR 7, which causes only part of a COMMON block to be shifted. ERROR 7 can result only from a COMMON statement that requires two doubleword variables to be stored an odd distance apart. (If an EQUIVALENCE statement requires this, the EQUIVALENCE is deleted--ERROR 6.) ERROR 7 is detected during COMMON statement processing, before EQUIVALENCE is even attempted. ERROR 7 produces the message

WARNING: THE { DOUBLE PRECISION / COMPLEX / DOUBLE COMPLEX } VARIABLE 'D' IN { BLANK COMMON / COMMON BLOCK /B/ } MUST ....
LIE ON A DOUBLEWORD BOUNDARY.
IT AND EVERYTHING BEYOND IT WILL BE SHIFTED UP BY ONE WORD.

Note that for this error to occur, D will not be the first doubleword variable in the COMMON block: some other doubleword variable must have preceded it to establish a parity preference.

Naturally, both ERROR 7 and ERROR 11 may occur in the same COMMON block, and ERROR 7 may occur more than once.

3.8.8 Finish Blank COMMON FINBLCOM _ ALLOCATE)

This routine exists mainly to check whether blank COMMON has overflowed. If it has, we complain with ERROR 9.

We also break up the 6-word item on the BLCOMLST, storing its results in BLCOMOVF, BLCOMPTC, BLCOMPTY, and BLCOMSZ (temps).

3.8.9 Finish Labeled COMMON FINLBCOM _ ALLOCATE

This does the same thing as Finish Blank COMMON except that it ignores temps. Overflow diangosis is done separately on each COMMON block.

3.8.10 Allocation Diagnostics Demonstration

The following several pages are extracted from an NSS memo describing Allocation Diagnostics and their objectives. Examples are given of all the basically different types of messages that may be produced.

The diagnostics are produced by compiling a FORTRAN source deck with some cleverly designed errors in it. This source deck is available in the Xerox Program Library, as described in the memo.

Subject: Sigma 5/7 FORTRAN IV Allocation Diagnostics Demo
National Software Support, Applied Programming
Bulletin 69-20-33.

3.9 Subprograms

3.9.1 Classification

There are several different ways that an identifier can be classified as a subprogram. The first two constitute definitions, the next two are references, and the last can be either.

1. Appearance in a FUNCTION, SUBROUTINE, or ENTRY statement. Of course, within a FUNCTION the name used as a scalar variable in which to return the result (a SUBROUTINE or ENTRY name should not be used within), but if the subprogram is internal, then the name is a subprogram outside. This means it has the SPROG class. It also has the SPROGDEF trait, since it is defined.

2. Defined as a statement function. Here again, the name has both the SPROG class and the SPROGDEF trait.

3. Appearance in a CALL or CONNECT statement. This registers the SPROG class, but not the SPROGDEF trait. This leaves open the question of whether the subprogram is defined within the program. It may turn out to be a later statement function or internal subprogram. (Actually, it is probably not very meaningful to CALL or CONNECT to a statement function, but the compiler will allow it.)

4. Referenced in an expression, followed by a left parenthesis. Assuming that the name has not been DIMENSIONed (which would make it an array) or appeared in a COMMON or EQUIVALENCE statement or the like (which would set the permissibility to not allow subprogram), a name that appears in an expression followed by a left parenthesis is assumed to be a function reference. This also registers the class SPROG. The compiler makes no particular distincition between SUBROUTINE names and function names, except that a function must have a type (Integer, Real, etc.), whereas a SUBROUTINE usually does not (it could be given one, but to no particular purpose).

5. Declared in an EXTERNAL statement. This immediately sets up the SPROG class. It also sets up the EXTERNAL trait and permissibility, whose effects depend on what the name later (or earlier) turns out to be. EXTERNAL on a SPROGDEF (name defined in the program) acts like an external DEF (i.e. makes the name available to other outside programs). EXTERNAL on an outside reference, on the other hand, has no particular effect. It does, however, assure that the name is a subprogram, without actually defining or referencing it. This is useful if you want the name to appear alone (without arguments) as an arguemnt to another subprogram. Since it is not followed by a left parenthesis, it would otherwise be assumed to be a scalar. The EXTERNAL declaration avoids this. The other use of EXTERNAL is to change an intrinsic function to a basic external one.

Like other symbols, subprogram names are kept on the symbol Table. They have the class SPROG and various other traits to characterize them. Subprograms

that have been identified as intrinsic functions have special information in two of the words in the SYMTABLE entry. (See Section 1.5 under SYMTABLE. Also "Intrinsic Functions", Section 3.6, "Process If Intrinsic", Section 3.10.1.3.5.3)

When Pass 1 scans a subprogram definition, it sends the name of the subprogram to Pass 2 and it also sends a Dummy Group Plex (DMYGRPX) identifying the dummies associated with this subprogram definition.

When Pass 1 scans a subprogram reference, it sends a Function Plex (FUNCPLEX), which is made up of two things: the subprogram name (it does not really have to be a function; this applies to CALLed SUBROUTINEs as well); and an Argument Group Plex (ARGPLEX) identifying the arguments that are to be passed on this particular reference. Obviously (and unlike deinfitions), there may be several references, each with different arguments. If the subprogram is an intrinsic function, the number and kind of arguments are restricted somewhat and diagnosed if violated. E.g. ALOG does not permit a complex argument; no intrinsic function permits a statement number or long alphanumeric string argument.

3.9.2 Generation

Pass 2 gnerates in-line code for the intrinsic functions. Where it can (for MAX, MOD, INT for instance), it gens code to evaluate such functions directly. The mathematical functions such as SIN, LOG, and SQRT are done by library subroutines; Pass 2 simply builds BALs to the appropriate subroutines for these functions. This code generation is done by Intrinsic Sprog Call Gen (INTRINSG, see 3.6.2) in Pass 2, which knows and makes use of some algebraic properties of intrinsic functions -- such as the fact the ABS and COS are even functions, and SIN and CONJG are odd functions.

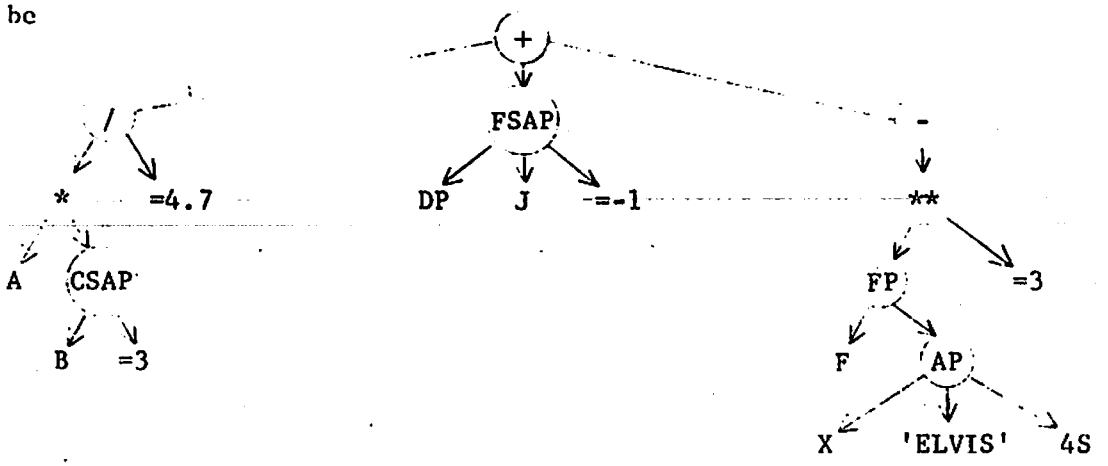
For each CALL statement or reference to a non-intrinsic function (including Statement Function), Pass 2 generates a Standard Calling Sequence (see 2.25) which consists of a BAL to the named function or subroutine, followed by a list of the arguments. For a description of how this code is generated, see 3.28.2.1 (Sprog Call Gen). For each SUBROUTINE, FUNCTION, or ENTRY statement, and for each Statement Function definition, Pass 2 also generates a Standard Receiving Sequence; this is a BAL to a library subroutine (called a Setup routine), followed by the list of dummies. The setup routine establishes the correspondence of arguments and dummies. Following the receiving sequence code, Pass 2 generates Prologue Code: code which establishes values needed to address elements of dummy arrays (see 3.78.2, SUBROUTINE Statement Gen).

3.10 Arithmetic Expressions

As discussed in Section 1.10.1, expressions are represented by plexes. For example, the diagramed plex representation for the expression

A*B(3)/4.7 + DP(J-1) - F(X,'ELVIS',4S)**3

would be



where: CSAP means Constant Subscripted Array Plex

FSAP means Fully Subscripted Array Plex

FP means Function Plex

AP means Argument Group Plex

On the PLEXLIST, this is represented by a group of plexes pointing at each other and at operands on other lists (e.g. identifiers, constants). Each plex begins with a total word count, created automatically when the plex was built. The listing below is sort of a symbolic representation of a possible PLEXLIST for the above expression. It differs from an actual PLEXLIST in several ways: the counts and locations are in decimal instead of hexadecimal; most of the fields are identified symbolically, rather than hexadecimally; no traits are shown; and the various plex groups would not actually come out in the precise order shown. None the less, the basic structure is correct. Each word has an address field that is either a count, a pointer to another location on the PLEXLIST, or a pointer to an operand on another list.

Word	Plex or Pointer	Mode	Address
0	count		5
1	SUMPLEX	D	traits
2	DIVPLEX	R	5 (on PLEXLIST)
3	FULSUBPX	D	17 (on PLEXLIST)
4	MINUSPLX	R	22 (on PLEXLIST)
5	count		4

```
6       DIVPLEX             R       traits
7       PRODPLEX        R       9 (on PLEXLIST)
8       RDCONLST        R       =4.7

9       count                   4
10      PRODPLEX        R       traits
11      SCALAR              R       A
12      CONSUBPX        R       13 (on PLEXLIST)

13      count                   4
14      CONSUBPX        R       traits
15      ARRAY               R       B
16      ICONSLST        I       =3

17      count                   5
18      FULSUBPX        D       traits
19      ARRAY               D       DP
20      SCALAR              I       J
21      ICONSLST        I       =-1

22      count                   3
23      MINUSPLX        R       traits
24      EXPPLEX             R       25 (on PLEXLIST)

25      count                   4
26      EXPPLEX             R       traits
27      FUNCPLEX        R       29 (on PLEXLIST)
28      ICONSLST        I       =3
```

29	count			4
30	FUNCPLEX	R		traits
31	SPROG		R	F
32	ARGPLEX		-	33 (on PLEXLIST)
33	count			5
34	ARGPLEX		-	traits
35	SCALAR		R	X
36	SCONSLST	-		='ELVIS'
37	LOCLBLST	-		4S

This plex structure is built up by the scanning routines in Pass 1. Pass 2 then processes this to generate code.

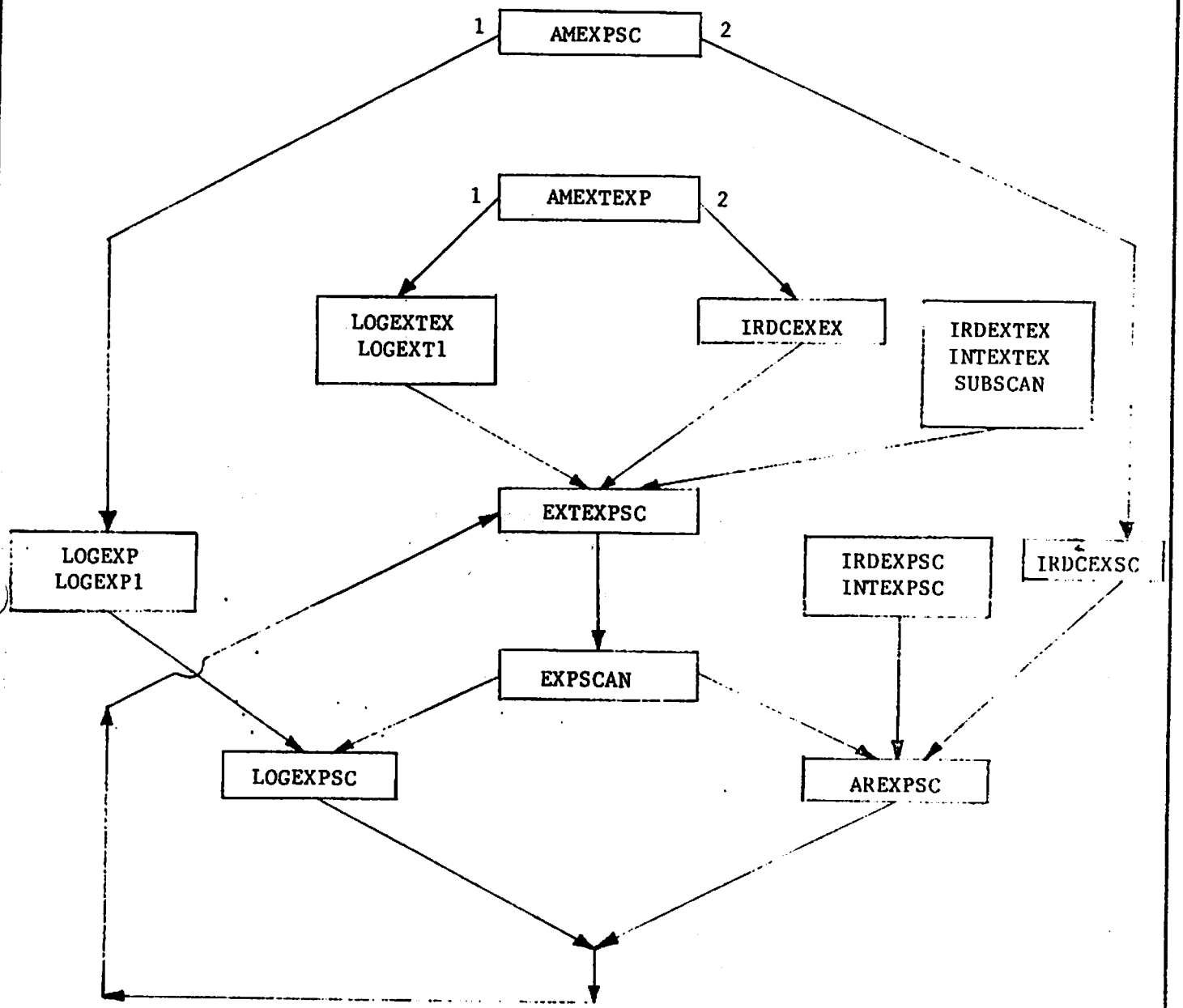
3.10.1 Arithmetic Expressions Pass 1)

3.10.1.1 Entry Points

There are a variety of entry points to expression scan, depending on the modes that are permissible for the expression. Some of these entry points are used only by expression scan itself when it recurses because of a nested subexpression. Also, there are some entry points for "extended expressions", which are assignments embedded in a statement. This was a standard feature in 9300 FORTRAN. It was more or less taken out of the Sigma FORTRAN, but unfortunately it was not as carefully taken out of the compiler as it was from the reference manual. Thus some users discovered that the feature was there and suggested that it be documented, especially since its use when not enclosed in parentheses was hghly inconsistent: in some places it was allowed, in others it was not. To be more consistent, and to avoid language ambiguities (such as REWIND J = N), we officially documented that extended expressions may appear only within parentheses. The compiler has not yet been changed to disallow the other uses, but it should be. Throughout the compiler (with the exception of the recursive call within parentheses), all the references to extended expression entries should be replaced with calls on the corresponding non-extended expression scan entry.

The various entries call each other as shown in the following diagram. Three of the calls are double calls, labeled 1 and 2. This is where either logical or arithmetic expressions are allowed, so we have to TRY logical first, then

scan for arithmetic if logical fails. All of the actual scanning is done in the two routines at the bottom, LOGEXPSC and AREXPSC.



The meanings of those entry points associated with arithmetic expressions are given below. (The logical ones are in Section 3.11.1.1)

AMEXPSC Any Mode Expression Scan. TRYs LOGEXP first. If that fails, it allows any arithmetic mode by calling IRDCEXSC.

IRDCEXSC Integer-Real-Double-Complex-(Komplex) Expression Scan. Calls AREXPSC with the I-R-D-C-K bits set in P0.

IRDEXPSC Integer-Real-Double Expression Scan. Calls AREXPSC with the I-R-D bits set in P0.

INTEXPSC Integer Expression Scan. Same as IRDEXPSC. This does not really restrict the expression to integer. All of the places that allow an integer expression also allow the other two modes, which are then truncated to integer.

AREXPSC Arithmetic Expression Scan. This is the routine that does the work. It expects mode permission bits in P0. Each element in the expression will then be checked against these bits by CHEKTYPE.

EXPSCAN Expression Scan. Also expects permission bits in P0, but allows them to specify logical. If they do, it calls LOGEXPSC, otherwise it calls AREXPSC. This is not a TRY. This is used by extended assignments (and other replacements) that already have their permissible modes set up (e.g., by the variable on the left of the equal sign). Note that this takes advantage of the knowledge that the logical and arithmetic bits will never both be set.

EXTEXPSC Extended Expression Scan. Expects the mode bits set in P0. Calls EXPSCAN and, if the resultant expression is only a variable of some sort (i.e., begins with a letter and is addressable), then it allows an equal sign and another expression (which also may be a variable and another equal sign and so on). This is not a real TRY. It is a much more "bottom up", and therefore less time consuming, retroactive test of what it found in EXPSCAN.

IRDEXTEX Integer-Real-Double Extended Expression. Calls EXTEXPSC with the I-R-D bits set in P0. All calls on this should be replaced with calls on IRDEXPSC.

INTEXTEX Integer Extended Expression. Same as IRDEXTEX. All calls on this should be replaced with calls on INTEXPSC.

SUBSCAN Subscript Scan. Same as IRDEXTEX. The calls on this can be left alone (the callers should not know what a subscript looks like), but the name should be transferred over to the routine IREDEXPSC/INTEXPSC.

IRDCEXEX Integer-Real-Double-Complex-(Komplex) Extended Expression. Calls EXTEXPSC with the I-R-D-C-K bits set in P0. All calls on this should be replaced with calls on IRDCEXSC.

AMEXTEXP Any Mode Extended Expression. TRYs LOGEXTEX first. If that fails, it allows any arithmetic modes by calling IRDCEXEX. All calls on this should be replaced by calls on AMEXPSC.

3.10.1.2 Hierarchy

An arithemetic expression is one or more terms separated by plus or minus. (There is a little added complication because these operators can also be unary.) A term is one or more factors separated by asterisk or slash. (There is a little complication here because an asterisk may also be the first part of an exponentiation operator.) A factor would be one or more elements separated by double asterisks, if exponentiation were processed left to right like the other operators (and as it was in older FORTRANs). Since exponentiation is processed right to left, however, a factor is either an element or an element raised to the power of another factor. (This is a recursive, but not circular, definition.) An element is an operand (e.g., scalar, array element, constant, etc.) or is an extended expression enclosed in parentheses.

3.10.1.2.1 Arithmetic Expression Scan AREXPSC

We get one term either from Term Scan (TERMSCAN) or Signed Term Scan (SGNTRMSC), depending on whether or not there is a leading sign on the term. Then if there is no plus or minus sign following that term, it is the only term and we can exit. The reserving and releasing of the PRODLIST is done for TERMSCAN (see below).

If there is more than one term, we reserve the SUMLIST and pile up the terms there. When SGNTRMSC comes back FALSE, that indicates no more terms, so we build a SUMPLEX (variable number of operands) from them.

3.10.1.2.2 Signed Term Scan SGNTRMSC)

If theree is no plus or minus sign next, we exit FALSE, indicating that ther is not another term here. Note that in the case of the first term, there may indeed be a term that does not begin with a sign. That is why AREXPSC calls TERMSCAN if SGNTRMSC comes back FALSE on the first term; there does have to be at least one term in an arithmetic expression or there is no expression.

If there is a plus sign, we just call TERMSCAN and return with what it gets. If there is a minus sign, we call TERMSCAN and then call Negate Term (NEGTERM) to negate it. This means if there is already a minus plex attached to the

term; it is removed (two minuses make a plus), if there is no minus plex attached to the term, one is attached.

3.10.1.2.3 Term Scan (TERMSCAN)

Strictly speaking, TERMSCAN should be looking for factors separated by multiply and divide signs. But because it saved some code, and was sort of cute, the looking for asterisks has been doubled up in Factor Scan (FACTORSC) with the looking for double asterisks (exponentiation). So we call FACTORSC to get the first factor (it cannot be preceded by a unary asterisk or slash), and FACTORSC tells us whether there is one asterisk following that factor. If not, we check for slash, exit if not, or call FACTORSC again and build a DIVPLEX between the first factor and the second. This then becomes a new factor, and we continue as we did after getting the first factor (i.e., we are told whether there was one asterisk following this second factor, and if not we check for slash again, and so on).

Whenever we do find a single trailing asterisk, we put the previous factor on the PRODLIST and keep calling FACTORSC and piling up factors on the PRODLIST as long as there are separating asterisks. When an asterisk does not follow a factor, we build a PRODPLEX (variable number of items) out of the factors, and then continue again as if we had just gotten one factor without an asterisk after it. In other words, we check for slash, and this may produce a DIVPLEX and then go back and check for more asterisks. Note that the PRODPLEX is a variable-size plex, for which we collect factors on the PRODLIST. On the other hand, the DIVPLEX is strictly a two-item plex. In a case like

A * B * C * D / E / F * G * H

the first four factors are collected and built into a PRODPLEX. Then this PRODPLEX is combined with E in a DIVPLEX. This DIVPLEX is combined with F to make another DIVPLEX. This second DIVPLEX is then combined with G and H in a three-item PRODPLEX.

A single expression may have several terms, each with PRODPLEXes that need to be built. Rather than repeatedly reserving and releasing the PRODLIST in TERMSCAN, we do it once, up in AREXPSC, knowing that TERMSCAN will leave it clean when it exits and knowing also that if we recurse around to TERMSCAN again, AREXPSC will have again reserved the PRODLIST at the new level.

3.10.1.2.4 Factor Scan (FACTORSC)

First we get an element, by calling Element Scan (ELEMSCAN). Is there an asterisk after it? If not, rather than just exit FALSE, we want to check for implicit multiplication (a missing asterisk). If an element is followed by a letter, a digit, or a left parenthesis, it can only mean that an operator is missing (or some other more horrendous mistake). So we flag the character

with a warning and proceed as if there had been a single asterisk, i.e., exit TRUE.

If there was an asterisk, we then check for a second one. If there is only one, we exit TRUE. If there are two, this is an exponentiation operator, and by the new rules of FORTRAN, this must be processed right to left. This means that the first element must be raised to the power of the whole factor to the right, so we recurse directly back into FACTORSC, build an EXPPLEX, and return with the answer of whether the second factor was followed by a single asterisk. Note that the second element may in turn be followed by a double asterisk, for example

A ** B ** C

In this case the third recursive level gets C, the second combines this with the B that it had gotten, and the first level combines this whole thing (representing B**C) with the A that it had gotten. In other words, the left hand elements get piled up on the WORKLIST, one per level, and then combined backwards.

It is a trivial change, noted in the listing, to make this process exponentiation left to right instead. One simply has to combine the new elements immediately as they are obtained, rather than recursing down.

3.10.1.2.5 Element Scan ELEMSCAN

This is used by both arithmetic and logical expressions. The basic operands of an expression are essentially the same in either case. They may be:

A scalar variable

A subscripted array

A subscripted multiple dummy

A function reference

A constant

An expression of the same kind, enclosed in parentheses

There is a set of permission bits in PO that indicate what modes of operand may appear in this expression. (For function arguments or subscripts, of course, a new permission word is recursively added to the bottom of the PARAMLST.) So after ELEMSCAN gets an element, it exits through CHEKTYPE, which compares the type of the element with the permission bits and gives a fatal delayed error if they do not match. Thus even in an arithmetic expression, ELEMSCAN will recognize an element such as .TRUE., but it will be daignosed by CHEKTYPE.

ELEMSCAN follows these steps:

1. (ELEMSCAN). If the first character is a letter, go to step 6 to scan for an identifier.

2. Otherwise, is the first character a left parenthesis? If not, the element can only be a constant (first character a digit or decimal point), so go to Constant Scan (CONSTSCN).

3. Element begins with left parenthesis. We have to TRY for complex constant, since its syntax is so ambiguous with nested expressions. However, we can avoid doing the TRY if the first character after the left parenthesis is not a digit, decimal point, plus sign, minus sign, or comma. (Comma is included because of the possibility of the user having omitted the real part of a complex constant. However, I do not think CONSTSCN is prepared to do anything about this, so it probably accomplishes nothing.) If the TRY is done, and is successful, we go off to CHEKTYPE to check the type and exit.

4. Element begins with left parenthesis, but is not complex constant. It must be an expression, therefore. So we call expression scan, then check for the right parenthesis following it. If the right parenthesis is there, great, we exit. (We do not have to go through CHEKTYPE, because the lower level did it when it got the actual operands.)

5. If the right parenthesis is not there, it may just be a missing right parenthesis, but only if the next character is something reasonable like a comma or semicolon or end-of-line. If it is one of these, we can give a missing right parenthesis warning; otherwise, we have to go to Illegal Syntax.

6. (IOC2). Element begins with a letter. Call IDSCAN to get an identifier.

7. If the identifier is followed by a left parenthesis, go to step 9; it is probably an array or function.

8. Identifier not followed by left parenthesis. This should be a scalar, which is handled by Scalar Scan (SCALARSC), below. However, if scalar is not permissible (see "Permissibility", Section 2.8.1), then we have some kind of erroneous situation, like an unsubscripted array name or a function with no arguments, which will be taken care of by their respective scans. So we can just fall into step 9, as if there had been a left parenthesis; the fact there is not will be picked up later.

9. (IOC3). Identifier is followed by left parenthesis (or erroneous situation as described in step 8). If the identifier is registered as ILLEGAL (see Section 3.76.1), go to ILLEGID (Section 3.10.1.3.7).

10. If the identifier is permissible to be a subprogram (SOP SPROG), go to Function Scan (FUNCSCAN, Section 3.10.1.3.5). Note that most functions appear without ever having been "declared" to be a subprograms (e.g., with

EXTERNAL or as a statement function definition). What is important is whether the identifier can be registered as a subprogram. Thus an unknown name will become a function, not an array; arrays must be dimensioned first.

11. The identifier is probably an array or multiple dummy. Before scanning its subscripts, we want its type (see REGTYPE, Section 2.8.3.1) and then check it for validity, so that any error message will come out marked on the identifier. If we did the CHEKTYPE at the end, as with other elements, the error would be flagged on the final right parenthesis, which would not be desirable.

12. If the identifier is permitted to become an array, go to Array Scan (ARRAYSCN, Section 3.10.1.3.2). It may not have been DIMENSIONed yet, which is an error, but it must have appeared in something like a COMMON statement to prevent it from being a subprogram, so it is probably an undimensioned array.

13. If the identifier is a multiple dummy, go to Multiple Dummy Scan (MULDMYSC, Section 3.10.1.3.4).

14. Only erroneous situation remain. The identifier may be a scalar followed by a left parenthesis, or it may be a UFO (Unidentifiable FORTRAN Object) which will produce UFO (Unexecutable FORTRAN Object) code. This can only result from something like a hardware error, that produces a meaningless Symbol Table entry. So we go to UFOERROR and then try to recover at BADIOREC (see Section 3.10.1.3.7).

3.10.1.3 Elements

3.10.1.3.1 Scalar Scan (SCALARSC)

The identifier has already been determined to be permissible to become a scalar, so we register it as one and set up the new permissibility for this. Then we go off to Register Type (REGTYPE) and Check Type (CHEKTYPE).

3.10.1.3.2 Array Scan (ARRAYSCN)

If the identifier is not already registered as an array, this is an error, which will be handled a little later. If it is an array, but it is not followed by a left parenthesis, this is also an error, though less serious. We can print a diagnostic and call Address First Element (ADRISTEL), which will assume all the subscripts were their lower bounds (usually one).

In the normal array case, we want to call Subscript Sequence Scan to get the subscripts. Then we want to call Combine Subscripts (COMBSCRS) to multiply all the subscripts together with the appropriate dimensions and come up with a

Constant Subscripted Array Plex or a Fully Subscripted Array Plex. In the process of doing this, COMBSCRS will put out diagnostics if there are too many or too few subscripts. (It does not know this until it compares them with the correct subscript count, which is off in a special entry on the GROUPLST, pointed to by an entry on the ARRAYLST, pointed to by the array's SYMTABLE entry. See Section 3.2.) Such a diagnostic has to be marked somewhere, and the array name is the best place. (The final right parenthesis would be easier, but could be confusing if the subscripts were very complicated, with nested expressions and subscripted subscripts and lots of right parentheses.) So we pick up the character count at the end of the identifier and store it in ARYSCTMP. However, we must not do this before we scan the subscripts, because they too might be arrays and would want to store a count in that temp. The temp would not be recursive, so we make it recursive by keeping the count on the WORKLIST until after the subscripts have been scanned, then storing it in the temp.

If the identifier was not an array, but is followed by a left parenthesis (and we already know it is neither a function nor a multiple dummy), then it may have been intended to be any array, appeared in EQUIVALENCE or something that prevented it from being classified as a function, but was never dimensioned. We go off to NOTARRAY to handle this (see Section 3.10.1.3.7).

If the identifier was not an array, and is also not followed by a left parenthesis, this should be an impossible situation. Is there any combination of uses that will result in an identifier that cannot be a scalar or a subprogram, is not a multiple dummy, but can be an array, but isn't? We do not know of any, so we print the UFO error and go off to Bad ID Recovery.

3.10.1.3.3 Subscript Sequence Scan (SCRSEQSC)

Subscript Sequence Scan (SCRSEQSC) calls SUBSCAN as many times as three are expressions separated by commas. SUBSCAN is currently the same as Integer Extended Expression, but should be changed to be the same as Integer Expression Scan (see Section 3.10.1.1). The subscript pointers are piled up on the SCREXLST (Script Expression List), which was first reserved. We do not process these subscripts; we just leave them on the SCREXLST for Combine Subscripts (COMBSCRS) to handle.

3.10.1.3.4 Multiple Dummy Scan (MULDMYSC)

To be a multiple dummy, the identifier must have explicitly appeared as such in a dummy list. There is no confusion (such as with arrays and functions) about whether or not something should be a multiple dummy. If a left parenthesis does not follow the name, we substitute a subscript of one (i.e., the first element of the multiple dummy) and output a warning diagnostic. Otherwise we call SUBSCAN once (a multiple dummy has only one subscript) and then require a right parenthesis. Then we go off to Process Multiple Dummy Subscript (PMULDSUB), which is a special entry to Combine Subscripts. It will

take our subscript expression (or the integer constant one that we created) and combine it with an artificial dimension of one, and come up with a subscript very much like that of an ordinary array.

3.10.1.3.5 Function Scan FUNCSCAN)

There ought to be a left parenthesis after a function name. If the parenthesis is missing, we put out a warning and proceed. Argument Sequence Scan (ARGSEQSC) is called next, and it is quite happy with no arguments (since it is also used by the CALL statement, which allows zero arguments). ARGSEQSC returns an ARGPLEX, which we then combine with the function name into a FUNCPLEX, and we exit with that in WO.

3.10.1.3.5.1 Argument Sequence Scan ARGSEQSC

Since most functions first appear without ever having been declared as subprograms, we want to register that class. It could have been done above in Function Scan, but this commons it up with the CALL statement.

Scanning for arguments is quite different if the function is intrinsic, so we call Process If Intrinsic (PRIFINTR), which handles the whole jub in that case. Otherwise, it returns FALSE and we continue. Also, if the function is not intrinsic, we can register and check its type; intrinsic functions do not use REGTYPE, because they have an inherent (built-in) type, rather than using the implicit (IJKLMN) rules. And we can set up the permissibility for EXTERNAL, which is the same as that for SPROG except that it disallows intrinsic so that next time we see the function, we will not have to search the Intrinsic List to see if it is intrinsic.

Now we reserve the ARGLIST and pile up argument pointers there. These are scanned by ARGSCAN, below. Then we build an ARGPLEX from these. If there was no left parenthesis, the ARGPLEX will be empty, specifying "no arguments".

3.10.1.3.5.2 Argument Scan ARGSCAN

An argument (to an ordinary function, not to an intrinsic) may be any of the following:

An array name (unsubscripted)

A subprogram name (with no arguments)

A statement number (Xerox or IBM form)

An expression (of any type)

Some of these are partially ambiguous with one another and unfortunately have to be handled by TRYs. A more "bottom up" approach that did not require this would be better. This is only partially "bottom up"; we will at least decide what we can based on the first character.

If the character is a letter, the argument is either an expression, an array, or a subporgram name standing alone. If we call expression scan first, it will not like the names standing alone and will give warning messages that are not appropriate here. So we have to TRY the names alone first. If it is not that, then we go off to get an expression (hopefully not an extended expression) of any mode.

If the first character is an ampersand, the argument must be an IBM - type statement number (e.g. &16), so we go off to Active Statement Number Scan (ACSTNOSC) to get the statement number and indicate that it has been potentially referenced (i.e., is "active").

If the first character is a digit, there is considerable ambiguity that has to be resolved by TRYs. The argument might be any of the following:

16HTHIS IS A STRING

16S or 16$

16

16*K+8

The first of two of those will not be accepted as expressions, so have to be TRYed separately. (Actually they could be TRYed both at once. The reason they are not is that the statement number scan needs to scan the integer and then look for the S or $, while the long Hollerith constant is handled altogether by Constant Scan.) If both of those TRYs fail, then we go off to get an expression.

If the first character is a quote sign, the argument may be (better be) the other form of long alphanumeric string (surrounded by quotes). This also is handled by Constant Scan, via the special entry point GETCONA that allows long alphanumeric strings.

If the first charcter is not a letter, digit, ampersand, or quote, then the argument must be an expression (e.g., beginning with plus, minus, decimal point, or left parenthesis).

3.10.1.3.5.2.1 Constant Argument Scan CONSARSC

This routine exists primarily to separate long Hollerith string arguments (beginning nH) from the other types of constant that will be allowed as parts of expressions. It is called with a TRY. It in turn calls GETCONA, the entry

point in Constant Scan that allows long Hollerith string constants. In the process, of course, GETCONA will allow many other types of constant as well. This is all right, except in certain circumstances. If GETCONA starts off on something like

```
16*K+8
```

it will scan the integer constant 16 all right, but it will not be prepared to combine this with the rest of the expression; this should be done by Term Scan, down in the expression hierarchy. So, if GETCONA succeeds, the constant it found must be followed by a comma, right parenthesis, or end-of-statement (which includes semi-colon). Otherwise, despite the fact that we have successfully scanned a whole constant, we have to fail, go through expression scan instead, and end up rescanning the same constant again. This is one of the unfortunate examples of "top down" inefficiency (see Section 2.7.3).

Since the first character is know to be a digit, it is certain that Constant Scan will, in fact, succeed in finding something, if only the one digit integer constant that that digit represents. In the course of scanning a lot of arguments in a program and doing a lot of these TRYs, we could accumulate a large number of meaningless integer constants on the ICONSLST. To prevent this, we save this list initially, to be restored if the TRY fails. Of course, this still allows for finding some other mode of constant that will not be wiped out, but this is not nearly so frequent.

3.10.1.3.5.2.2 Name Alone Argument Scan (NAMARGSC)

As in the previous example, this routine will always find some sort of name, since a letter has been seen. Therefore, we save the Symbol Table first. The name must be permissible to be a name alone argument (in general this means it must be an array or subprogram), and it must be followed by a comma, right parenthesis, or end. If so, then we can set up new permissibility for NAMEARG. As a slight digression, I cannot reconstruct any reason for this permissibility. In order to be a name argument, the identifier must already be an array or a subprogram, and the NAMEARG permissibility only adds intrinsic to the restricted list. But how could something already be known to be a subprogram but still allow intrinsic, without being intrinsic? And if it is already intrinsic, it would not allow name argument. Probably this instruction could be removed safely, but as it is it does not hurt anything.

If the name is a subprogram, we are done. However, if it is an array, it may be the first appearance of the array, in which case we have to register its implicit type via REGTYPE, which will also set up the element size for the array. For a subprogram, we do not really have to know the type; if it does not already have one, we will pass it as an argument with no type. This is, in fact, what we must do with SUBROUTINEs, which generally do not have a type. But with arrays, we have to pass the address of the first element, which we cannot properly determine without knowing the type. Also, the correct type must be passed in the calling sequence word.

Finally, with array names, we go off to Address First Element (ADRISTEL), a part of Combine Subscripts, which will compute the proper offset to build a subscripted array plex pointing at the first element.

3.10.1.3.5.2.3 Statement Number Argument Scan STNOARSC

First we save the two label lists (since this routine is TRYed). Then we scan for an integer. It can be followed by either S or $ or, just to allow the user wide variety, by both, in either order (which is the same as just $). Then we go off to register either a local or non-local label with an active reference (RELCLREF or RGNLCREF).

3.10.1.3.5.3 Process If Intrinsic (PRIFINTR)

This routine has the responsibility of dealing with intrinsic and basic external functions. As such, there are several situations it can encounter:

1. The function can be intrinsic, but has not been used yet (this the first appearance). Then if the name is not one of those on the INTRILST (a static list identifying all known intrinsic functions), we do nothing and exit FALSE.

2. Same but the name is on the intrinsic list. The entry on the INTRILST gives us two pieces of information. One is the function's inherent type (e.g., DSIN is Double Precision, not the Real that you would expect from the IJKLMN rule). The identifier must inherit this type; this is done by Inherit Type (INHTYPE, Section 2.8.3.4). However, if Automatic Double Precision (ADP) is in effect, we have to do some finagling. If the inherent type is Real or Complex, we have to convert it to Double Precision or Double Complex, so that Pass 2 will generate appropriate Double Precision coding (which may involve calling a different non-standard evaluator).

   The second thing we keep on the INTRILST is information about what type and how many arguments each intrinsic function may have. Since there are only six categories, we do this by having six routines (F1 through F6) for processing intrinsic arguments. Then the INTRILST contains merely the address of the appropriate argument processing routine. For example, the SQRT entry on the Intrinsic List says F1, which allows exactly one argument of Integer, Real, or Double Precision mode. This address (e.g., of F1) is moved into the last word of the symbol's entry on the Symbol Table, so it will be there from now on when we have to process this function.

   Two other things have to be don thee on first appearance of an intrinsic function. 1) Send the function's code number to Pass 2 by storing it in the Symbol Tale entry's fourth word. The code number is determined by the

function's position in the INTRILST. 2) Register the identifier with the INTRNSIC trait and set up the corresponding permissibility.

Now we can fall into case 3, below, which is when the function has already appeared and is known to be intrinsic.

3. The function has already been established as intrinsic. Its type has been set up, so we call CHEKTYPE to check it. (If it is invalid, we want to print the diagnostic here, rather than after the arguments.) Then we reserve the ARGLIST in preparation for piling up arguments. If there is no left parenthesis following an intrinsic function, we print the "too few arguments" diagnostic (a fatal delayed error) and skip the argument scanning. Otherwise, we get (from the sixth word of the SYMTABLE entry) the address of the proper argument processing routine and go there. This routine will demand the right number and mode of arguments and will build an ARGPLEX from them, for use by Function Scan. If the number of arguments is incorrect, it will print the "too few" or "too many" arguments message, which will cause a delayed fail. But it goes ahead and builds the ARGPLEX from the wrong number of arguments.

4. The function cannot be intrinsic (permissibility bit missing), but it can be basic external. This usually results from the name being declared EXTERNAL, which is recommended to users as the way to convert an intrinsic function to a basic external (so that, for example, he can substitute his own version). In this case, we are only interested in the function's inherent type, not its arguments (there are no special restrictions on the arguments of a basic external function). So we still have to search the INTRILST to see if the function is there (there is no distinction on this list between intrinsic and basic external functions; a basic external function is just an intrinsic function that has been declared EXTERNAL). If it is there, we inherit its type into the Symbol Table (again taking into account ADP mode). Also we set up new permissibility for BASICEXT which, among other things, does not allow basic external! This is because the only thing being basic external does for a function is to get it a special type, and the function already has its special type, so there is no point in searching the INTRILST again next time.

5. The function cannot be intrinsic (which also means that it <u>is</u> not intrinsic) and it also cannot be basic external (which might mean that it already is basic external or, among other things, that it has been given an explicit type. Declaring an explicit type also makes it pointless to be basic external. In other words, it is either not permissible or not necessary to search the INTRILST for this name in order to give it a special type. In this case, we just exit FALSE, to tell Argument Sequence Scan that we did <u>not</u> process the arguments.

3.10.1.3.6 Constant Scan CONSTSCN

The standard entry point to Constant Scan (namely CONSTSCN) is used for normal appearances in expressions and does three things:

1. Calls SCANCONS, which will scan any type of constant except long Hollerith (longer than four characters) and long hexadecimal (longer than eight digits) and leave the value in CENTRAL and a type code in CONSTYPE.

2. Calls REGCONST, which picks up the type code from CONSTYPE and goes off to the appropriate constant registering routine. This routine will register the constant on one of the various constant lists (see Section 3.7.1) and return a pointer.

3. Exits via CHEKTYPE, which assures that this type of constant is permissible in this expression. If not, it prints an error message.

There are some other entry points used when we want to allow long Hollerith or hexadecimal and/or when we do not want to check the type (because we are not in an expression). These entry points are:

GETCONA -- Used by the CONNECT statement, the PAUSE and STOP statements, and by Constant Argument Scan (Section 3.10.1.3.5.2.1), to allow the quote form of Hollerith constant (which should not really be called a Hollerith constant; it should be called a string or alphanumeric constant) but not the long hexadecimal.

GETCONB -- Used by the DATA statement, which wants to allow everything.
GETCONST -- Used by SINCOLM1 (in-line assembly), which wants to disallow long hexadeciaml and either allow or disallow long string constants, depending on whether the constant is preceded by a plus or minus sign. (String constants are not supposed to be signed.) This GETCONST is not related to the one used by the DATA statement in STASCAN.

These special entries also have to be prepared to scan complex constants (enclosed in parentheses), whereas in expressions this is handled at a higher level, by TRYing CPXCONST. We do not want to TRY anything, because we know we want a constant and once we have seen the left parenthesis, we know it must be a complex constant. We cannot JRS to CPXCONST instead of using TRY, because CPXCONST may decide to set the automatic succeed flag on the TRYLIST, and if we have not entered with TRY, there is no such flag on the TRYLIST. So we use a TRY, but we enter CPXCONST one instruction early (at DCPXCON, Demand Complex Constant) and immediately set the automatic succeed flag, so that the TRY will work like a JRS (see Section 2.7.4).

For further information on constant scanning, see Section 3.7.2.

3.10.1.3.7 Identifier Errors

These fall into several categories:

1. NOTARRAY. An identifier is followed by a left parenthesis. The identifier cannot be a subprogram, it can be be an array, but it is not. This usually results from the user having forgotten to DIMENSION an array. Unfortunately, some of the time he does this, the name will automatically get classified as a function, which will not give him the diagnostic. He may not find the problem until he notices the strange external subprogram required. There is nothing we can do about this; it is part of the FORTRAN definition. If by luck, however, the name has appeared in a COMMON or EQUIVALENCE statement for example, this disallows subprogram. Then we can be pretty sure that this was supposed to be an array and has not been DIMENSIONed, so we put out a message to that effect. Note that if the name was already known to be a scalar, then we will treat this as implicit multiplication.

2. ILLEGID. The SOT class of the identifier is ILLEGAL. This is done by Statement Function Definition, to prevent recursive calling of such functions. This automatically produces a message about "REFLEXIVE DEFINITION".

3. UFOERROR, BADIDREC. A name absolutely cannot be classified. Either it has an impossible SOT class (from a hardware error or unknown software bug) or it has that peculiar set of characteristics discussed in Array Scan (Section 3.10.1.3.2) wherein a name cannot be a scalar or subprogram (because of permissibility), is not a multiple dummy, can be an array, but is not. This should not be a possible situation (COMMON or EQUIVALENCE would allow scalar as well as array), and so is also probably a hardware error. In this case we do two things. We print the error message

   INEXPLICABLE ERROR: PROBABLE HARDWARE FAILURE. SEE OPERATINS MANUAL.

   UFOCODE= .

   where the UFOCODE is the location from which UFOERROR was called. This may help the systems programmer track down at least who gave the message.

   Then we want to try to recover. There is a left parenthesis next, which is probably followed by something that was meant to be either a group of subscripts, a group of arguments, or a nested sub-expression. We do not know which, but we would like to scan as much as possible, in order to detect possible additional errors. If we scan it as an expression, or as a group of subscripts, it will not like things like 'ABCDE' or 16S, which he might have intended as arguments. Fortunately, if we scan this as a group of arguments, that includes all the other possibilities as subsets. So that is what we do. We call ARGSEQSC, which scans arguments and produces an ARGPLEX, which we then throw away. The statement is going to fail anyway, so why carry these things around? Also why try to build a FUNCPLEX with them, when the identifier they follow is not a function?

3.10.2 Arithmetic Expression Generation Pass 2)

Arithmetic expression generation is not hard to understand, but it seems hard to understand because there are so many entry points to it. There are variations dependent on whether you want the result to come out in memory, or a register, or both; there are variations dependent on whether you object to having the result come out with the wrong sign; and there are endless variations on the theme: What Do We Do About The Mode. The names for the entry points are discussed in 2.16.

The SIGNLIST and the MODELIST are involved in all of this. The SIGNLIST bottom contains the sign of a current subexpression (i.e., indicates whether it has been generated with the wrong sign (TRUE) or not (FALSE)). The MODELIST bottom indicates what mode the expression result is desired in. These lists are used by the lower level routines that generate code for the various operators.

The main distinction among entry points has to do with Addressability. A thing is Addressable if it is in memory and can be used as the address of a generated instruction. The main question is: do you want the result in memory or in a register? An addressable quantity (such as 3 or X) is already in memory, so calling Grntee Adressable generates no code. But to guarantee A+B addressable requires that you add A and B and store the result in a temp. If A+B is desired in a register, no store is done into a temp. The Grntee Addressable entry points return a pointer to the expression or temp in WD, while the Grntee In AC entries do not.

The Replacement Plex (REPLPLEX) comes in for some special treatment. The code generator was designed so that a replacement (=) operator could appear anywhere in an expression. If a Replacement Plex appears at a Grntee Addressable entry, we go off and generate it and use the left-hand variable as the resultant address. (This assumes there is no problem with mode.) In the Grntee In AC entries, it makes a difference whether the replacement can be handled by a memory-to-memory instruction (as J = J+1 can be handled by MTW,1). In this case the result is in memory but not in a register, so it has to be fetched. Memory-to-memory assignments are generated by Special Replacment Gen (SPECREPG) which returns an answer telling whether or not it could do the replacement. If SPECREPG cannot handle it, we call Ordinary Replacment Gen (ORDREPG), knowing that the result will be in the accumulator as well as memory.

In Grntee Addressable, the strategy for replacements is to generate it and use the left-hand variable as the resultant address. For Grntee Addressable And In AC, we use the left-hand variable as the resultant address, but we may have to do a fetch from it beforehand. This happens if Special Replacement Gen has succeeded and generated a memory-to-memory instruction.

After the various entry points have dealt with addressability and the sign and mode problems, any non-trivial expression will have to go through Gen By Plex Type (PLEXGEN), whose operation is explained in 3.10.2.1.

3.10.2.1 PLEXGEN Gen By Plex Type _ Pass 2 __

PLEXGEN is a routine through which most arithmetic expression generation must eventually pass. Any arithmetic operator other than minus (-) comes through here--specifically: sum (+), product (*), quotient (/), exponentiation (**), mode switch, function call, and replacement (=).

The entry conventions are as follows. A plex pointer is in W0. Code to evaluate the expression represented by the pointer is generated onto the CODELIST and the pointer is discarded from W0. The code generated causes the expression result to be left in an accumulator. If the expression is of Integer mode, the result is left in the register specified by RTEMP; of other than Integer, the register is the Standard Accumulator for that mode. Every element in the expression is converted to the mode specified on the bottom of the MODELIST (exceptions are mentioned later under the Complex Mixture Trait). The expression is allowed to be calculated with the wrong sign, and if so, the bottom word of the SIGNLIST has its parity changed. The size of the SIGNLIST and MODELIST remain the same, while the WORKLIST is decreased by one.

PLEXGEN simply checks the plex pointer in W0 to make sure that it is one of those for which it can generate code, then branches through a table to one of the seven appropriate generation routines. The generation routines are: SUMGEN, PRODGEN, QUOTGEN, EXPONGEN, MODSWG, FCALLGEN, and REPACGEN. An eigth type of plex pointer is in the acceptable range (the Extended Relational Plex), but it is not acceptable for generation and causes a JRS to P2ABORT. For use of this plex, see Simplify Logical Expression.

3.10.2.2 SUMGEN and PRODGEN Pass 2 _

Sums and products are the only arithmetic operators that can have a variable number of operands. This is one reason tha these two generators are rather long and involved. Another reason is the Complex Mixture Flag (CPXMFLAG). A further reason for complication in products is that Integer products are treated specially so that multiplication by powers of 2 can be generated by shifts.

Ignoring the integer product business for the moment, both SUMGEN and PRODGEN start off with a QJS to SUMPRODS whose long name is "Sum Product Setup and Is Term Complex Mixed?". The basic purpose of this routine dates back to the SDS 920 FORTRAN II Compiler; in those days, nothing was Complex, much less Complex Mixed. The routine decides which term (subexpression) of the sum or product would be the best one to start evaluation with. For example, consider the sum W + X + Y*Z. It is best to start by evaluating Y*Z. If we started by adding W and X, we would have to store in a temp before we could evaluate Y*Z (remember: this compiler is largely ignorant of multiple accumulators). So the 920 version of the routine reserved the TERMLIST and pulled the plex to it, then examined the terms to see if it could find a non-addressable one, and left it on the bottom of the TERMLIST. It further went on to evaluate that first term and throw away its pointer.

The basic idea is the same in the Sigma version, but it is complicated by the concept of Complex Mixture. Though this feature does cause better code to be generated in some cases, the cost is rather exorbitant. But it is there and needs to be understood. The basic thought is: if there are Complex and non-Complex terms in an expression, it is a pity to promote all the non-Complex terms to Complex and evaluate them with Complex arithmetic when we know the imaginary parts are going to be zero. We can, with no loss of accuracy, evaluate them in Real or Double and convert them just before combining them with anything Complex.

The thought was: in a sum of, say, seven terms, we are allowed to evaluate the terms in any order we choose. (If the user wants to control the order, let him use parentheses.) So let us examine all the terms and find all the non-Complex ones and evaluate them first and add them with non-Complex arithmetic, then convert to Complex and add in the other terms.

A further thought occurred: most sums and products are <u>not</u> composed of both Complex and non-Complex terms, i.e., are not complex mixtures. So let us find a quick way of distinguishing them. The solution is the Complex Mixture Flag, a trait that is specially handled by the plex-building POPs to tell whether this particular mode of the expression is Complex Mixed. (This logic occurs on all plex-building, but it is in machine language and therefore not very expensive.)

SUMPRODS checks for Complex Mixture and, if not, goes quickly to Find First Term (FINDTERM) and exits FALSE. If this is a Complex Mixture, SUMPRODS reserves a file on the Complex Term List (CPXTMLST) and puts all the complex (or Komplex) terms there. It then moves a few files around so that the Complex terms are in a file on the TERMLIST, then the non-Complex terms are in another file on the TERMLIST. (The CPXTMLST is empty, as is the WORKLIST.) We set the mode to Real or Double (depending on the whole expression's being Complex or Komplex) and leave this mode appended to the bottom of the MODELIST. Then we JRS to FINDTERM (which will operate only on the bottom file of the TERMLIST--the non-Complex file), then exit TRUE.

So SUMPRODS returns with one or two files on the TERMLIST and an answer FALSE or TRUE, respectively. It already has generated code for the first term and has appended its sign to the SIGNLIST. In the TRUE case, it has also appended a non-Complex mode to the MODELIST. The rest of the calling routine (which is SUMGEN or PRODGEN) must generate the rest of the terms and Exclusive OR the sign parity into the next-to-bottom entry on the SIGNLIST. In the Complex Mixture case, it must finish generating the bottom file on the term list, then generate a mode conversion to Complex or Komplex, then generate the terms in the other file.

To generate the other terms of a sum or product, both routines use a subroutine called GTADSOPR (Grntee Operand Signed Addressable). This routine, whose design also dates from the SDS 920, is rather sneaky. If the operand is signed addressable, it merely appends the sign to the bottom of the SIGNLIST and exits. But if it is not addressable, it stores the current result in a temp, then evaluates the new expression into AC and returns the temp pointer

as the addressable thing. In other words, it interchanges the two operands. It also interchanges the two signs on the bottom of the signlist. For example, in SUMGEN if we are attempting to generate A*B + C*D and we have finished C*D and have a minus sign on the bottom of the SIGNLIST if we then JRS to GTADSOPR with the A*B pointer in W0, the compiler will generate a store into a temp, a load of A and a multiply by B, and will return with the temp pointer in W0 and two signs on the bottom of the SIGNLIST: minus on the bottom for the temp and plus in the next word up for the result in AC. SUMGEN then decides whether to build an Add or Subtract instruction by whether the two bottom signs are the same or not. It then discards the bottom sign, since the AC sign is the result sign. This same sequence is repeated for the other terms of the sum. It is done in a routine called Partial Sum Gen which keeps looping until it has exhausted the terms on the TERMLIST. In the Complex Mixture case, Partial Sum Gen is called once, then a mode conversion generated, and there is another call on Partial Sum Gen. Finally, we Update Sign And Exit (UPSIGNEX), which consists of Exclusive ORing the bottom sign into the next sign up and discarding the bottom.

The generation of the other terms of a product is quite similar to sum. Here we do not compare signs (as Sum did to determine Add or Subtract); we just generate Multiply instructions and keep Exclusive ORing the signs. There is a Partial Product Gen that may have to be called twice to generate the non-Complex and Complex parts of the expression.

3.10.2.3 Quotient Gen DIVGEN _ Pass 2 _

When there is no Complex Mixture, quotients are easy to generate. We pull the plex to the WORKLIST (knowing there are two terms). We evaluate the denominator using GTADSBYM (Grntee Signed Addressable By Mode), then the numerator, using GTAGSBYM (Grntee Signed in AC By Mode). Then we BAM a Divide instruction. The two "Signed" calls on the expression evaluator have left two signs on the SIGNLIST. We fall into Update Two Signs and exit.

If the SBIT option is set, and the expression mode is INTEGER, we insist on correct signs in both the denominator and the numerator. This option insures "correct" integer results even in overflow cases.

The Complex Mixture case is more trouble. Each operand is evaluated, if necessary, by GTACSCPO (Grntee Operand Complex Signed In AC). This routine expects a Complex or Komplex mode on the bottom of the MODELIST. If the expression is non-Complex and non-addressable, we set the MODELIST to Real or Double and evaluate it in that mode, then convert to Complex or Komplex. Then we join the main flow of the code and BAM a divide, then update two signs and exit.

GTACSCPO checks for a Function Call Plex. If TRUE, we go directly ot PLEXGEN, not setting the mode to Real or Double. The idea is to avoid generating two mode conversions.

3.10.2.4 Expon Gen EXPONGEN _ Pass 2 _

The ** operator is somewhat unusual in the way it handles the question of the mode of its right operand. If the right-hand operand is non-integer, it is evaluated in the mode of the whole expression in which the operation is embedded. (This is the same convention used for +, -, *, and /.) If the right-hand operand is integer, however, it is evaluated with integer arithmetic, regardless of the mode of the left operand or the rest of the expression. There are two sets of "power" routines (for raising to a power): the ones that raise something to an integer power and those that raise something to a power of the same mode as the left operand. Thus the following libarary names are used:

```
9PWRII

9PWRRI          9PWRRR

9PWRDI          9PWRDD

9PWRCI          9PWRCC

9PWRKI          9PWRKK
```

and Expon Gen must figure out which of these to call, if any.

In certain cases, no routine gets called at all: this occurs if the exponent is an integer constant of value 2 or 3 (for square or cube). For these cases, multiplication instructions are generated in-line. The compiler exercises some sign logic in the square and cube cases. It always allows the left operand to be computed with the wrong sign. If a square is being generated, it forgets the sign because (-X)**2 = (+X)**2. In the cube case, where (-X)**3 = -(+X)**3, the sign is "postponed". This means that the sign is entered on the bottom of the SIGNLIST, and the outer expression decides what it wants to do about it. Generation of the left expression has left an extra sign appended to the bottom of the SIGNLIST. The cube case exits through UPSIGNEX (Update Sign and Exit) which consist of

```
MOF         SIGNLIST

EOS         *SIGNBOT

BRL         EXITML
```

thus removing the bottom sign by exclusive ORing it into the next sign up. The square case could exit through DSIGNEX (Discard Sign and Exit), which consists of

```
MMO       SIGNBOT

BRL       EXITML
```

But, as it happens, the square case does the two above POPs in-line.

If the exponent is not 2 or 3, there are two basic cases: the Power XI case (something raised to an integer power) and the Power XX case (something raised to its own mode power). These cases select not only which set of library routines is involved, but also what kind of load is generated for the second operand. In the Power XI case, the right operand is loaded into register 11, 9, 11, 11, or 13, respectively, as the left operand is Integer, Real, Double, Complex, or Komplex. Odd numbered registers were chosen for these operands because integer multiplication and division can be done correctly in them. ISee the X rox Sigma Computer reference manual to find out why these operations are inconvenient in even-numbered registers.) In fact, the compiler never generates these operations in these registers. If the integer operand is complicated, we evaluate it in Register 9 and store it into a temp, then load it into the appropriate register. If somebody wanted to do it, he could modify the compiler to do a better job on complicated Integer exponents. The integer accumulator can be changed by altering RTEMP. It might involve the addition of 10 or 15 POPs, and there would still be cases that have to be evaluated in register 9: those involving function calls or exponentiation. It is probably not worth the trouble.

The only remaining case is the Power XX case. Here we guarantee the right operand addressable in the mode of the expression (not allowing the wrong sign). Then we guarantee the left operand in the accumulator (again with right sign), then load the right-hand operand into the appropriate register or registers, then generate the appropriate library call by mode. In this case, the second operand is a word in register 11, a word in register 10, a doubleword in 10, a doubleword in 10, or a quadruple word in 12, depending on whether the operand is Integer, Real, Double, Complex, or Komplex. When we are done with this generation there are no extra signs hanging around on the SIGNLIST because we have not allowed wrong signs. (This statement is also true of the Power XI case, except for squares and cubes.) So we just exit.

The Power X1 and Power XX cases involve a lot of mode indexing. This kind of thing is done regularly throughout arithmetic expression generation (for example, the compiler uses mode indexing to decide whether to generate DW, FDS, or FDL instructions), but in EXPONGEN the mode indexing is rather messy. The BAM POP gets its use. This POP is in fact an Execute POP indexed by the mode of W0. Its name, Build by Addrss Mode, comes from the fact that it usually controls a BIF POP, which gets its address part from W0. Here BAM sometimes executes a table of ADR POPs that control later generation.

The subroutine LOAD2XI (Load 2nd Operand as for X**I) needs mode indexing, but the mode it needs cannot be in W0. Therefore, instead of using BAM, it BRLs to WM0DX1ML (W0 Mode to X1 ML), then gets the thing it wants into W0 and uses an indexed XEQ POP to do its generating.

3.10.2.5 Mode Switch Gen MODSWG _ Pass 2 _

The name "Mode Switch Gen" (MODSWG) should not be confused with "Mode Conversion Gen" (MODCVGEN), which generates all the mode conversion code needed in Pass 2. Mode Switch Gen generates code for a special plex called a Mode Switch Plex and calls upon Mode Conversion Gen to do some of its work.

The Mode Switch Plex allows specific control of the evaluation mode of certain parts of an expression. In the general case, there are three modes involved: the mode of the outer expression, the mode of the Mode Switch Plex, and the mode of the inner (operand) expression. Mode Switch Gen generates the operand expression in its own mode (demanding correct sign), then converts to the mode of the plex, and then to the mode of the outer expression.

The Mode Switch Plex is actually of little use because the mode-promoting rules of the compiler handle nearly everything that needs handling. It is used in two cases: non-Integer subscripts, and relational operators whose operands have different modes. Suppose an array element has non-Integer subscripts, say A(X+Y,J,Z). Let X and Y and Z be real, and J integer. Then we want to evaluate X+Y in floating point, then convert the result ot Integer, convert Z to Integer, then combine the three subscripts with multiplies and adds in Integer. Combine Subscripts, which is used in Pass 1, attaches a FIXPLEX to every subscript expression that is not integer. The FIXPLEX is the same as the INTGPLEX and is the integer version of the Mode Switch Plex. The other versions are called SNGLPLEX, DOUBPLEX, CMPXPLEX and KMPXPLEX.

The only other use of the Mode Switch Plex (and hence later of MODSWG) is in Simplify Relational (SIMRELAT) which is part of Simplify Logical Expression. In a relational operation, the modes of the expressions must agree in order for a compare instruction to work. So if the modes disagree, one or both operands get Mode Switch Plexes built onto them. The only case where both operands would get plexes is where one is complex and the other double (they would both be promoted to komplex).

3.10.2.6 Function Call Gen FCALLGEN _ Pass 2 _

Function Call plexes are refreshingly easy to generate because all the work is done a lower level routine, Sprog Call Gen (SPCALLGN), which is described in 3.28.4. FCALLGEN simply JRSs to SPCALLGN, then checks the subprogram name pointer in WO to see if it is logical. If not, then it is arithmetic in mode and may require mode conversion, for which we BRU to Mode Conversion Gen (MODCVGEN). If the mode is logical, we exit through CLR1EXML (Clear 1 and Exit ML).

3.10.2.7 Replacement Gen REPACGEN _ Pass 2 _

REPACGEN is called by PLEXGEN when it has to evaluate a Replacement Plex (REPLPLEX). The function of REPACGEN (Replacement into Accumulator Gen) is to generate code which evaluates the "right hand side" expression, stores it in one or more variables (performing any necessary mode conversions), and also leaves it in the accumulator by the mode on Mode List bottom. The actual work is done by one of three subroutines: Special Replacement Gen (SPECREPG), Ordinary Replacement Gen (ORDREPG), or Logical Replacement Gen (LOGREPG). If the Replacment Plex is of Logical mode, then REPACGEN calls LOGREPG (see 3.10.2.7.1). Otherwise, it first calls SPECREPG to see if the replacement is of the form V=V+I. If so, it generates the corresponding code (see 3.10.2.7.2) and returns the answer TRUE; REPACGEN then BRUs to FECHSMOD (Fetch and Set Mode) to generate the load and possible mode conversion. If SPECREPG returns FALSE (indicating that the replacement could not be done via "add to memory"), then REPACGEN calls ORDREPG (see 3.10.2.7.3) to generate the expression evaluation and stores, then BRUs to MODCVGEN (Mode Conversion Gen) to gen the final mode conversion, if necessary. Each of the three Replacement Gen routines returns a pointer to the last variable stored into, so that the calling routine knows what mode the value was left in. REPACGEN passes this pointer along to FECHSMOD or MODCVGEN, which discards it.

3.10.2.7.1 Logical Replacement Gen LOGREPG _ Pass 2)

This subroutine is used by Replacement into Accumulator Gen (REPACGEN) and Evaluate Logical Expression (EVALOGEX) to handle logical Replacement Plex evaluation. LOGREPG does a Reserve and PUL to place the Replacement Plex constituents (one or more logical pointers -- the "left hand side" - and a logical expression pointer) on the Replacement List (REPLLIST). It then copies the logical expression pointer to the worklist and calls EVALOGEX (Evaluate Logical Expression) to generate the evaluation code. Next, it obtains a copy of one of the variables (the last one to be stored into) to leave on the worklist upon exiting, as an address pointer (this poinfer will be returned to the caller of GTADULOG in case the latter calls LOGREPG). LOGREPG then generates a logical store into each of the left-hand-side variables. It also sets the Clobber Index Trigger (CLOBXTRG) as an indication that index register quantities should not be remembered beyond the current statement (stores into variables may, in general, affect the values of subscript expressions being remembered).

3.10.2.7.2 Special Replacement Gen SPECREPG Pass 2) This routine is used by GTADUOWN (via General Replacement Gen, GREPLGEN), GTBOUBYM, and REPACGEN. It tries to make use of MTW or AWM instructions to simplify and speed up replacements of the form V=V+I where V is an integer variable. Thus, for the statements J=J+2 and M=M+J/K it generates

```
MTW,2     J
```

and

```
LW,9 J

DW,9 K

AWM,9    N
```

respectively. The conditions under which code like this can be generated are:

1. the thing to be evaluated must be an integer Replacement Plex;

2. the "right hand side" expression must be of integer type;

3. there must be only one left-hand-side variable;

4. the right-hand-side expression must be a sum (SUMPLEX), one of whose terms (at least) is the left-hand-side variable.

If all of these conditions are met, then the replacement is of the desired form V=V+I, where I is the sum of the remaining terms. To test these conditions, SPECREPG proceeds as follows: (1) a SOM and a SOT determine that it is an integer Replacement Plex; (2) the plex is PULled to the Replacement List and its terms are counted: there must be exactly two (one left-side variable and the right-side expression); (3) the right-side expression is then tested (via SOM and SOT) to assure that it is an integer Sum Plex; (4) its terms are PULled to the Term List, and are there searched (via SER) to make sure that V is among them. We now form an expression pointer for I, the reaminder of the right-side expression, as follows: the V term is removed from the collection of right-side terms by moving the last term to the position V occupied, and shortening the list by one term; if only one term remains, then that term <u>is</u> the expression I; if more than one term remains, they are collected (via PLO and CIC) into a Sum Plex, the pointer to which is used for I. Now, if I is not a constant, we generate code to evaluate I, followed by AWM,9 V. If I <u>is</u> a constant, we call upon BILDINCR (Build Increment) to generate "add to V" code, according to the range in which I's value lies:

1. if i=0, no code is generated;

2. if -8 -i-+7, we gen
```
MTW,i    V
```

3. if -2 -i-2 -1, we gen
```
LI,9 i

AWM,9    V
```

4. if <u>i</u> does not fall into any of the above special ranges, we gen
```
LW,9 =i

AWM,9    V
```

Most of the above assumes that the replacement was of the special form (in which case, code is generated as indicated, a pointer to V is returned in WO, and the answer is TRUE); if it was not, then SPECREPG removes anything it may have left on lists during the tests, generates no code, and returns with the answer FALSE. In either case, it sets the Clobber Index Trigger.

3.10.2.7.3 Ordinary Replacement Gen ORDREPG _ Pass 2)

ORDREPG PULs the Replacement Plex constituents to the Replacement List, calls GTACUREM to generate expression evaluation code, then, for each left-side variable, calls MODCVGEN to generate mode conversion code and generates a store into the variable. At each such state, we keep track of the current expression mode on the bottom of the Mode List. After the last store has been generated, we MOF the last Mode List item into WO as the address pointer to be returned (this is a pointer to the last variable stored into, and therefore has the current expression mode). ORDREPG sets the Clobber Index Trigger.

3.11 Logical Expressions

The plex structure for logical expressions is very much like that for arithmetic expressions, except that some different plex names are used. These are: EORPLEX, ORPLEX, ANDPLEX, NOTPLEX, GEPLEX, LEPLEX, LTPLEX, GTPLEX, EQPLEX, NEPLEX, and EXRELPLX (Extended Relational Plex). In addition, the relational plexes will contain arithmetic expression plexes as operands.. In Pass 2, some further plexes are created to facilitate generating better object code. For example, a LTPLEX (less than) that compares something with zero will be transformed into a LTOPLEX (less than zero), which will signal that the test can be made without a compare instruction.

3.11.1 Logical Expressions Pass 1)

As described in the section on "Top-Down Programming" (Section 2.7.3), logical expressions present quite a problem to a top-down method of scanning. It is not so much that logical expressions by themselves are any more complicated. They have their own hierarchy of operators that is straightforward to process. The problem is that there are several places where an expression about to be scanned may be either arithmetic or logical (e.g., IF statements, subprogram arguments), and a logical expression can contain an arithemtic expression as an operand in a relational. So consider the following four expressions that might appear in an IF statement or as an argument:

1. A*B

2. A*B .GE. X

3. L .OR. A*B .GE. X

4. L .OR. A

Number one is an arithemetic expression. Number two is a logical expression, but it starts off exactly like the arithmetic expression. If we try to scan this as an arithemtic expression first, it will succeed, but when we see the .GE. operator, we are not in the right place in the logical expression hierarchy to process it as one operand of a relational expression.

Number three is another logical expression that contains both logical and relational operators and operands. Number four is an error, since A is not a logical operand. However, if we were to scan it on the assumption that it must be the first operand of a relational expression, then we would get a syntax error on the following blank character (for not being a relational operator) and this would not be very meaningful to the user (who probably intended A to be logical and had no intention of writing a relational expression).

The upshot of all this is that we are forced to do a number of TRYs in scanning logical expressions, whenever it it is possible for either a logical or arithemetic expression to appear. We TRY logical expression before arithmetic, because a logical expression can contain an arithemetic as part of it. Furthermore, within logical expressions, we have to TRY relational expression first on every element. Thus, in a case like example four above, the TRY on A as a relational fails, so it is then scanned as a logical element and gets a diagnostic as not being the right type to be a logical element.

This produces some nice diagnostics in certain cases, but it is extremely time-consuming. An ordinary arithmetic expression, when it appears in an IF statement or as an argument, will cause two levels of TRY. The first level is logical expression, and the second is relational expression, and at the second level, the expression is completely scanned. Then, because relational expression fails, all this work is thrown out, both TRYs fail, and the expression is completely scanned again, producing all the same plexes and Symbol Table entries, constants, etc. This has to be one of the most inefficiently designed parts of the whole compiler. It resulted from carrying over the "top-down" philosophy of the 900 Series FORTRAN II (a much simpler language) without sufficient time spent analyzing the problems that would result from the new language features, such as logical expressions, relational expressions, complex constants, expressions in output lists. A more "bottom-up" approach would not only be much faster, but would probably produce error messages which are just as good if not better.

3.11.1.1 Entry Points

This is more or less a continuation of Section 3.10.1.1, the entry points for arithmetic expressions. The diagram there showed five entry points used only for logical expressions. These are described below.

LOGEXPSC Logical Expression Scan. This is the routine that does the real work for logical expressions. It processes the hierarchy of logical and relational operators. It expects, in P0, permission bits allowing only logical elements, so that Element Scan (which is shared by both arithmetic and logical expressions) can assure that all operands are logical.

LOGEXP Called with a TRY, by Any Mode Expression Scan. It saves several lists (EXPSAVE) and calls LOGEXPSC with the PERML (Permit Logical) permission word in P0 .

LOGEXP1 Same as LOGEXP, except it is called with a JRS instead of a TRY, so it does not save any lists. This is used by the REPEAT statement when the variable is logical. It knows it must have a logical expression, so it does not have to be TRYed. Note that in this case, the automatic succeed flag (on the bottom of the TRYLIST) may get set (there is always one there, even at the statement level), but it will have no effect.

LOGEXTEX Logical Extended Expression. Same as LOGEXP, but allows an extended expression. All calls on this (there is only one) should be replaced with calls on LOGEXP.

LOGEXT1 Same as LOGEXTEX, except called with a JRS instead of a TRY. This is used only by the Logical IF statement, which knows it wants a logical expression. This call should be replaced with a call on LOGEXP1.

3.11.1.2 Hierarchy

A logical expressiion is one or more OR-terms separated by .EOR. operators. An OR-term is one or more AND-terms separated by .OR. operators. An AND-term is one or more logical factors separated by .AND. operators. A logical factor is a logical element, which may or may not be preceded by a .NOT. operator. A logical element is either an operand like an arithmetic operand (variable, function, constant, nested expression, etc.), but of logical type, or it is a relational expression.

A relational expression is two or more arithmetic expressions separated by relational operators. If there are exactly two arithmetic expressions (one relational operator), this is a simple relational. If there are more, it is an extended relational. If any arithmetic expression is complex or double complex, it can be connected only by an .EQ. or .NE. operator.

3.11.1.2.1 Logical Expression Scan (LOGEXPSC)

This could also be called "EOR-term Scan" to be consistent with the following two sections. First we get one OR-term from ORTERMSC. If it is not followed by an .EOR. operator, we exit. The whole expression has been scanned.

If there is an .EOR. operator, we reserve the EORLIST and keep calling ORTERMSC, piling operands on the EORLIST as long as there are separating .EOR. operators. Then we build a multiple-word EORPLEX from these operands.

Also, once we have seen an .EOR. operator (or any other logical operator below), we set the Automatic Succeed Flag (on the TRYLIST) to indicate that this must be a logical expression. (See Section 2.7.4.)

3.11.1.2.2 OR-term Scan (ORTERMSC)

This is exactly the same as above, except that it calls ANDTRMSC for its operands, piles them on the ORLIST, and looks for .OR. operators separating them. Then it builds an ORPLEX.

3.11.1.2.3 AND-term Scan (ANDTRMSC)

Also exactly the same, but calls LOGFCTSC for operands, piles them on the ANDLIST, looks for .AND. operators, and builds an ANDPLEX.

3.11.1.2.4 Logical Factor Scan (LOGFCTSC)

If there is not a .NOT. operator first, then we simply go off to LOGELMSC, which gets a logical element and exits. If there is a .NOT. operator, we call LOGELMSC and then call Falsify Factor (FALSFACT). This builds a NOTPLEX unless there already is one, in which case it removes it (two .NOT.s cancel each other out).

If the first character is a ¬ (NOT sign), we print a warning and treat it as a .NOT. operator. Unfortunately the NOT sign does not print on most printers, which makes it look like we are treating a blank as a .NOT. operator. We are NOT.

3.11.1.2.5 Logical Element Scan (LOGELMSC)

This routine is deceptively simple. It TRYs RELEXPSC (Relational Expression Scan). If that fails, it goes off to ELEMSCAN (Element Scan), the same routine used by arithmetic expressions. Note, however, that the permission bits in PO say "logical", so the operands (variables, arrays, functions) will have to be of logical type or they will be diagnosed. Furthermore, if a nested subexpression comes along, it will also automatically be scanned as a

logical expression because of these permission bits. This will bring us right back down the above hierarchy.

I say this is deceptively simple, because it results in a large part of the extra scanning that occurs in logical expressions. That TRY for relational expression may do quite a lot of wasted work before it comes back FALSE.

3.11.1.3 Relational Expression Scan RELEXPSC

Since this is TRYed, we first save the usual "expression" lists. Then we reserve the WORKLIST (which is pretty unusual), in order to pile up arithmetic expressions and relational operators. We call IROCEXSC to get an arithmetic expression (Complex is all right; see below). Then we call RELOPEXP (Relational Operator and Expression), which tells us whether there is a relational operator and another arithmetic expression. If not, this is not a relational expression, so we fail, which means that LOGELMSC will go off to ELEMSCAN instead.

If there is an operator and expression, we then have to see whether this is an extended relational, i.e., is there another operator and expression? If not, this is a simple relational, so we take the plex that RELOPEXP returned to us and build it, connecting the two arithmetic operands. We exit with this.

If this is an extended relational, then we do not build the plex that was returned to us. We just leave it on the WORKLIST and keep calling RELOPEXP until it comes back FALSE. At this point, the bottom file on the WORKLIST looks something like this:

expression

plex type

expression

.

.

.

plex type

expression

Each "plex type" is the actual code word that we would normally FIP to build a plex for that operator. But for extended relationals, instead of building each individual plex, we leave them as shown above and build an EXRELPLX to tie the whole thing together, and we let Pass 2 worry about untangling it.

3.11.1.3.1 Relational Operator and Arithmetic Expression RELOPEXP)

This routine decides which relational operators are permissible here (if the previous expression was Complex/Komplex, the operator can only be .EQ. or .NE.) and then what mode of expression is permissible after the operator (if the operator is .GT., .LT., .GE., or .LE., the following expression cannot be Complex/Komplex).

Thus, if the previous expression was Complex/Komplex, we call RELOPSEN to get an equal or not equal operator, then call IRDCEXSC for the following expression, then exit TRUE. If no operator was found, we exit FALSE.

If the previous expression was Integer/Real/Double, then we first call RELOPSGL to look for a greater or less variety of operator. If it does not find one, then we proceed as with Complex by calling RELOPSEN. If it does find one, however, then we call IRDEXPSC to allow no Complex expressions, then exit TRUE. The TRUE answer tells RELEXPSC that we found an operator and another expression and have left them in that order on the WORKLIST.

3.11.1.3.2 Relational Operators RELOPSGL, RELOPSEN)

Each of these looks for certain of the realtional operators. RELOPSGL looks for .GE., .LE., .LT., , .GT., and . RELOPSEN looks for .EQ. and .NE.. If one of the appropriate operators is not found, we exit FALSE. If it is found, we return in WO the code word for the corresponding plex. For example, an .EQ. operator results in an EQPLEX on the WORKLIST. Then we do a couple of other things that have to do with the TRY mechanism. First we set the Automatic Succeed Flag for the TRY on Relational Expression Scan. This is kept on the bottom of the TRYLIST. This indicates that we have found a relational operator , so even if we subsequently fail, there is no point in having the TRY fail and going off and scanning for seomething else.

There is one further thing we can do. Consider the statement:

```
IF (A>B .AMD. L) GO TO 3
```

The Automatic Succeed Flag for relational expressions is not needed, because the relational expression succeeds anyway. Later, however, the logical expression fails, due to the misspelled .AND. operator. Never having seen a logical _____ _ operator, the Automatic Succeed Flag for Logical Expression Scan is not set, so Any Mode Expression Scan thinks it ought to go off and scan for an arithmetic expression (since the logical expression failed). This will produce a horrible diagnostic on the . We have to tell Logical Expression Scan that, because we found a relational operator, this thing must be a logical expression. The Automatic Succeed Flag we set will not do any good, because it only affects Relational Expression Scan. Logical Expression Scan cannot wait for Relational Expression Scan to finish successfully and then set its own Automatic Succeed Flag, because in a case like

```
IF (A>B> ***)  GO TO 3
```

Relational Expression Scan never does finish successfully. What has to be done is for the relational operator scans to be able to set the Automatic Succeed Flag for Logical Expression Scan, which is the next level of TRY up. This is a bit peculiar, but it is what we do. We reach up to the third word on the TRYLIST (knowing that the current TRYLIST entry is two words and the previous level's Automatic Succeed Flag is the bottom word on its entry) and set the next level's flag.

3.11.2 Logical Expression Generation Pass 2)

Unlike arithmetic expressions which are generated in one pass over the expression, logical expressions are done in two passes. The first is called Simplify Logical Expression (SIMPEXP) and receives a logical expression pointer in W0, tears the expression apart and puts it back together, and returns a new logical expression pointer in W0. Then for the actual code generation, there are three entry points: Logical Exp Gen Jump If True (EXPJUMPT), Logical Exp Gen Jump If False (EXPJUMPF), and Evaluate Logical Expression (EVALOGEX).

On the 9300 there was an additional generator especially designed for materializing logical answers (for use in logical assignment statements and/or subprogram arguments). It generated very clever code for this kind of situation. This was the first generator installed in the 9300 compiler, and the so-called Jumping Generator (see above) was not put in until some six months after initial delivery. For logical IF statements, we would materialize an answer in the A (or sometimes B) register, then test the answer and jump. This tended to produce rather horrible code for logical IF statements, although the logical assignment statements were great. Unfortunately, nobody seemed to write logical assignment statements, and customers were not too pleased with the code they got for logical IF.

On The Sigma there were several reasons for <u>not</u> doing an elaborate materializing generator: first, nobody seems to write logical assignment statements; second, the instruction set does not lend itself to such fiendishly clever code (on the 9300, Skip commands and COPY instructions were heavily used). So the lgocial expressions are generated with branching in mind, and if we need a materialized answer we branch to places that load register 9 with .TRUE. or .FALSE.. The only case where we are a little bit clever is where the expression on the right of the "=" sign is addressable. Here we do not test the variable and branch, but load it and store it. (If the same expression appears in a logical IF statement or a REPEAT WHILE statement, of course, we test it and branch.) Thus for the statements

```
LOGICAL P, Q

P = Q
```

we generate

```
        LW,9 Q

        STW,9     P
```

Logical answers are contained in the sign bit of the result and, knowing this, Simplify Logical Expression can do a few clever things so that terms that are not obviously addressable become addressable. For example, consider

```
        REAL X

        LOGICAL P

        P = X .LT. 0
```

Here the truth of the expression X.LT.0 is contained in the sign bit of X. So the compiler generates

```
        LW,9 X

        STW,9     P
```

and it can do this even if X is double precision.

Simplify Logical Expression is mainly concerned with simplifying relational operations. For one thing, we unwind extended relationals (such as A .LT. B .LE. C) into the .AND. of two or more relationals. For another, we check for relationals in which one operand is zero. For these we replace a dyadic plex with a monadic plex. For example, J .EQ. 0 comes in as a dyadic plex involving J and 0 (an .EQ. Plex) and goes out a monadic plex involving J (an .EQ. Zero Plex). There are thirteen possible plexes coming into Simplify Logical Expression and nineteen coming out. This is so because there are six additional plexes for the relationls with zero, one less plex because the Extended Relational Plex does not occur coming out of Simplify Logical Expression, and one more because the .EOR. operator is translated into two different plexes--the Easy .EOR. Plex and the Hard .EOR. Plex.

There is not much work in Simplify .AND. and Simplify .OR. . Their main function is to simplify their operands. But there is also some re-ordering of terms. Since the branching code for .AND. and .OR. can cause some of the terms to be skipped, we try to arrange things so that the easy terms are tested first (improving the probability that we may never have to test the hard terms). For example, the statements

```
        REAL X, Y, Z

        LOGICAL P

        IF (P .OR. X**2.GT.Y/Z**3 .OR.  Z.EQ.0) GO TO 23
```

we will generate the P term first, then the Z.EQ.0 term, and finally the ugly mess in the middle. Thus we get

```
        LW,9 P


        BLZ       23S
        LW,8 Z
        BEZ       23S
        LW,8 X
        FMS,8     X
        STW,8     1TEMP
        LW,8 Z
        FMS,8     Z
        FMS,8     Z
        STW,8     2TEMP
        LW,8 Y
        FDS,8     2TEMP
        CW,8 1TEMP
        BL        23S
```

The test used by Simplify .AND. and Simplify .OR. to determine whether a term is easy (and hence should be done early) or hard is the Test Addressable trait. This trait (TADRFLAG) indicates that this term can be tested in memory with a single instruction (such as MTW,0) and will leave its relation with zero in the condition codes. In fact for such a term we do not generate a MTW,0 instruction but we generate a Load instead because it is faster.

Simplify .EOR. is qyite different from .AND. and .OR. . For one thing, all the terms of an .EOR. must be tested in the generated code--none can be skipped. For another, we would like to combine all the addressable terms with the EOR instruction. This instruction leaves the condition codes properly set and is a good way to combine addressable terms. Simplify .EOR. separates the terms into Hard and Easy, aso do Simplify .AND. and Simplify .OR., but here the test for easiness is Addressability rather than Test Addressability. (The EOR instruction will operate on the sign bit.) Then if we have two or more easy terms, we combine them into an Easy .EOR. Plex (EEORPLEX); two or more hard terms form a Hard .EOR. Plex (HEORPLEX), and one of the terms of the hard

plex may be an easy plex. There is quite a bit of logic in SIMPEOR about whether ther are zero, one, or more of each kind of term. SIMPEOR also strips off (and counts) .NOT. operators from its terms; if there is an odd number of them, it builds a .NOT. plex onto the whole .EOR. result.

Code generation for the Easy .EOR. Plex is a snap. We just generate a LW,9 for the first term and EOR,9 for each other term, followed by BLZ for the Jump If True case, and BGEZ for the Jump If False. Generation for the Hard .EOR. Plex is hard. I mean hard, because both the generated code and the generator that generates it are hard to read. The code is efficient, but complicated. Each of the terms of the .EOR. will be generated twice except the first term. The generated code starts out testing the first term, and from there the code splits into two paths. These may be thought of as the "False" path and the "True" path. Every time we find a FALSE answer for one of the terms, we continue in the same path, because the FALSE answer does not change the answer of the whole .EOR. . Whenever we hit a TRUE answer, however, we jump over to the other path. Thus the "hard" terms are each tested twice (once in each path) during generation, but no term actually gets tested twice in execution. Perhaps an example will help. Consider

```
          If (A< B .EOR. C<D .EOR. E<F)  GO TO 23

                          LW,8    A
                          CW,8    B
                          BL      2G
      LW,8    C                          2G    LW,8    C
      CW,8    D                                CW,8    D
      BL      3G                               BL      1G
1G    LW,8    E                          3G    LW,8    E
      CW,8    F                                CW,8    F
      BL      23S                              BGE     23S
      B       4G                         4G    EQU     $
```

In memory, the right-hand column of code will follow the left. The code generator is difficult to follow because it generates the left-hand and right-hand paths of code at the same time and delays outputting of the second. Code for the second path is kept on the .EOR. Code List (EORCOLST).

Like all the jumping generators, Hard .EOR. has two entry points--one to jump if TRUE, the other to jump if FALSE. For HEORJMPT we ADR EXPJUMPT (Logical Expression Gen Jump If True) and branch to a common point; for HEORJMPF we ADR EXPJUMPF and fall into the same point. The only place that PO is used is in the initial branch (see the top of the above code example)--to determine whether that branch jumps if TRUE or FALSE. All the other branches jump if TRUE, except the last branch in the 2nd column, which always jumps if FALSE.

Generating code for .AND. and .OR. is quite easy, especially when compared to .EOR. . The easiest ones are .OR. Gen Jump If True and .AND. Gen Jump If False, and these are quite similar to each other. The .OR. Generator simply takes each operand in turn and generates it to jump if TRUE; the .AND. generator takes each one and has it jump if FALSE. (The label to be jumped to, by the way, is kept on the bottom of the Special Label List (SPLBLIST); this is true throughout the logical generator.) .AND. Gen Jump If True creates a label and puts it on the bottom of the SPLBLIST, then generates all but one of its terms to jump if FALSE to that label. Then it removes that label from the list and generates the last term to jump if TRUE. Finally it defines the created label-sometimes called the "jump-around" label. .OR. Gen Jump If False operates in an analogous way.

For relational operators, Simplify Logical Exp not only checks for zero constant operands but also decides which operand should be in the accumulator and which in memory when a comparison is made. The decision is on the basis of which operand is addressable, or, sometimes, which operand is constant. (The latter question matters because the LI instruction is faster than CI.) When we reach the generation stage, we know the right operand (WO) is to be in memory and the left in AC. We start by evaluating the right operand if it needs evaluating (i.e., if it is not addressable or of the wrong mode). In this evaluation, we allow the sign to come out wrong. (We do not allow a wrong sign in the case of an Integer expression when the compiler's SBIT option is set; if we did, results could be misleading.) If the sign comes out wrong, we force the left operand's sign to come out wrong also; then after the comparison, if the signs are wrong, we change the branch instruction. For example, a BG would become a BL , and so forth. This procedure gives us a chance of eliminating a negation.

All six relations are quite straightforward to generate. But there is a minor ugliness when it comes to double ccomplex operands. First of all, you are not allowed to use omplex or double complex operands with any relational operators except .EQ. and .NE. (there is no definition for one complex number being "less" than another). But in the double complex case there is a problem, since there is no quadruple-word compare instruction. So we do a semi-cowardly thing: we load the first operand into registers 8-11 and the second into 12-15. Then we do double-word compares of 8 with 12 and 10 with 14. Even this is not completely easy to do, since in building a Load Multiple, one has to generate an LCI before it (indicating how many words to load); if indexing is involved, the index register loads want to come before the LM but after the LCI and if this happens the condition codes get clobbered. So there is a little routine called Load Komplex 2nd Operand (LOKPXOP2) that is also used for quotients and powers. This falls into a routine called Do Komplex Thing (DOKPXTHG), which begins with a comment that explains waht is done to put things right.

3.12 Real Time

Specification of the Real Time Option (RT) causes the compiler to (1) generate reenterable SUBROUTINEs and FUNCTIONs, and (2) allow the CONNECT statement, by means of which the user can hook up a particular interrupt to a given FORTRAN subroutine.

In reentering a particular subroutine, it is necessary to save the context (local variables, temps, dummies) of that subroutine for its outer level of execution. The Real Time Temp (RTMP) aids the run-time in knowing when to save context, and where the context is. RTMP is located at the end of the Variable CSECT, following all the local variables, dummies, temps, and private temps. The right half of RTMP contains a count of the number of such context words (including RTMP); if n be this count, and rloc the location of RTMP, then the context block to be saved extends from rloc-n+1 to rloc. The left half of RTMP is used to contain a flag indicating whether the subroutine is currently in execution. Initially, this flag, called the "sentinel", is set to +1. Upon being entered, the subroutine sets it to -1. If, during the subroutine's execution, an interrupt occurs, causing the subroutine to be reentered, the negative sentinel value causes a library subroutine (9RSAVE) to be called: this routine saves the context for the FORTRAN subroutine being reentered. Upon RETURNing, the subroutine resets the sentinel to +1 and (if called via a CONNECT sequence) calls 9REXIT which restores the previous (outer) context, if any had been saved (i.e., if this is not the outermost level). In so doing, it restores the outer value of the sentinel (which was necessarily -1). Thus, the setting of the sentinel to +1 in the RETURN code takes effect only in two cases: (a) the subroutine was called with an ordinary CALL statement, rather then entered as the result of a CONNECTed interrupt; (b) we are RETURNing from the outermost level of a CONNECTed interrupt subroutine.

Note that the context region for a reentrant subroutine is the whole Variable DCECT for the compilation. If the compilation includes more than one subprogram, there is no way of separating context among them; it will all be saved whenever one is reentered. To prevent getting into trouble with this, only one reentrant subprogram per compilation is allowed, and then it is allowed only in a subprogram compilation (as opposed to a main program). The Real Time Sprog Trigger (RTSPGTRG) is set in SUBROUTINE statement gen only under the following conditions:

1. It is not a main program; i.e., MAINPTRG is reset.
2. The RT option was specified; i.e., RTTRIG is set.
3. This is the first SUBROUTINE definition; i.e., RTTEMPCT is zero.

It is RTSPGRTG, not RTTRIG, that controls the generation of reentrant SUBROUTINE code and the corresponding RETURN code. This code is as follows:

```
subr LCH,4      1RTMP          load/negate sentinel
```

```
BLEZ S+3            branch if first entrance
LI,4 1RTMP          reentrance: pass address
BAL,2    9RSAVE                 of RTMP to 9SAVE; push context
STH,4    1RTMP          set sentinel to -1
.
.
.    (ordinary SUBROUTINE prologue and body)
.
.
.
LI,4 1              reset sentinel to +1
STH,4    1RTMP
B        .linktemp  return
```

The first portion of the above code is generated by Real Time Sprog Head Gen (RTSPHDGN), called by Sprog Head Gen (SPHEADGN) when RTSPGTRG is set. The final code is produced by Real Time Exit Gen (RTEXITGN), called by the various RETURN gen rotuines if RTSPGTRG is set.

The call on 9REXIT is part of the sequence generated by CONNECT and executed upon return from the user's interrupt subroutine. See section 3.32.2 for a description thereof.

In Pass 3, if the Real Time Temp Count (RTTEMPCT) is nonzero (it can only be zero or one), Print Real Time Temps (PRRTTMPS) does the following:

1. Geneates an ORG (via BOORG) to the end of the V section (as given in RTEMPORG);

2. Computes the sentinel word = X'00010000'+n, where n is the total context size, given in VSIZE; this word is generated in binary output via a call on BODATUM;

3. If we are producing an object listing (LO), we print a line of the form

   ```
   1RTMP    DATA,2    1,n
   ```

   this appears as the last line under the heading "Local Variables".

3.13 Control Card Options

The FORTRAN control card options are scanned by FOPTIONS (see 4.3.3) at the beginning fo the first compilation (and possibly other compilations, depending on the BC option). Each of the nineteen options has a trigger associated with it; the name of the trigger is that of the option followed by TRIG (or TRG). With two exceptions, each trigger is set if and only if the corresponding option was specified. The exceptions are:

(1) If the compilation is being done on-line in UTS, GOTRIG (corresponding to the GO option) is set, whether or not the GO option was given;

(2) If the compilation is being done on-line in UTS, and if no options were given, then PSTRIG is set; thus, the default options for an on-line compilation are GO, PS.

Following is a description of the effects each option has on the compilation:

ADP Automatic Double Precision)

This option only affects Pass 1. When ADPTRIG is set, STASCAN interprets a REAL (or REAL*4) statement as DOUBLE PRECISION, and COMPLEX (or COMPLEX*8) as DOUBLE COMPLEX; EXSCAN changes the mode of an intrinsic function from R to D, C to K; CONSCAN does the same for the mode of constants; and PASS1ML initializes the Implicit Type Table so that letters other than I through N are associated with type DOUBLE PRECISION, instead of REAL.

ASA American Standards Assoc. Storage Allocation)

This is an option which was needed on the 9300 compiler, but not on the Sigma version; it is accepted as an option only for the sake of control card compatibility. There is a trigger (ASATRIG) associated with it, but it is not tested anywhere in the compiler.

BC Batch Compile)

There are two forms of this option:

BC

BC(n)

where n is a positive decimal integer. When either of these has been given, BCTRIG is set. There is an additional word (BCNUMBER) associated with this option: it contains n in the second case, -1 in the first. The effect BC has lies in two areas, both described elsewhere in detail: (1) in FOPTIONS to determine what to do at the beginning of each compilation (see FOPTIONS,

4.3.3), and (2) how to handle EOD and EOF conditions encountered while reading SI records (see DOREAD, 4.2.1.2).

BO (Binary Output)

BOTRIG is tested only by OUTBO. The various subroutines in OUTBO (see 2.28) all start by testing BOTRIG and GOTRIG; if neither is set, no binary output is to be produced, so the subroutines merely exit without doing anything. If either one of them is set, then binary object code is produced; if, furthermore, BOTRIG is set, then completed records are transmitted to M:BO via BOWRITE.

CDEB (Compiler Debug)

The checkout version of the compiler contained a debug package which was entered before the execution of each POP when CDEBTRIG was set. The test of CDEBTRIG was in the Interpret Loop in INTERPTR.

DEBUG (Run-Time Debug Interface)

This option causes the compiler to generate code to interface the FORTRAN program with the FORTRAN Debug Package (FDP) at run-time. During Pass 1, DEBUGTRG is tested at the following points:

(1) Label Field Scan (LBLFLDSC in PASS1F): to send source line numbers along to Pass 2;

(2) In STASCAN, at the beginning of a statement, to put a "Null Statment" on the Code List (see 3.19.1) in preparation for the following;

(3) In STASCAN, after an executable statement, to cover the "Null Statement" with an "Executable Statement" indicator;

(4) In FILRET (STASCAN), to force the generation of a RETURN or STOP before any SUBROUTINE, FUNCTION, or END statement;

(5) In FINISHUP (PASS1F), to force generation of a full Name List at the end of Pass 1.

During Pass 2, the DEBUG implications are too numerous to list here; see 3.19.1 for a discussion of the additional code generated when DEBUGTRG is set. Pass 3 tests DEBUGTRG only in Print Private Temps (PRPTMPS in PASS3F). In DEBUG mode, the first Private Temp generated is the Debug Chain Temp used by FDP. PRPTMPS lists and generates this temp as follows:

```
IPTMP    DATA 0        DEBUG CHAIN WORD
```

GO _____ Output to GO File)

This option trigger is set if the compiler is running on-line under UTS, whether or not the GO option was given on the control card. When running in batch mode, GOTRIG is set if and only if the GO option was given. GOTRIG, like BOTRIG, is tested only in OUTBO: it causes completed binary object code records to be directed to the GO file, via the subroutine GOWRITE.

LO List Object Code)

In Pass 1, the TRUE state of LOTRIG causes source lines and error lines to be sent along to Pass 3 for later inclusion with the Object Listing; this is done in SENDCARD, SENDERRS, and SENDPRBF in PASS1F. Pass 2 tests LOTRIG only in the CNL POP: if LOTRIG is FLASE, then CNL does nothing. At the beginning of Pass 3, if LOTRIG is set we do a page eject prior to producing the object listing. In Pass 3 there are many tests of LOTRIG, mainly in PASS3F and OUTPROG, to determine whether to build and print object listing lines, or skip the code that does this. The only other test is in SUMMARY, which causes a page eject before the summary if both an Object Listing and a Summary are requested.

LS List Source)

At the beginning of Pass 1 if LSTRIG is set, we do a page eject. LSTRIG is sensed in Pass 1 by SENDPRBF and SENDERBF: through these routines source and error lines are printed on the LO device. Pass 2 does not test LSTRIG. The only test in Pass 3 is in Summary, which does a page eject before the summary if both a Source Listing and Summary are requested.

NMP No Memory Protection

Tested only during Pass 3. PASS3F tests it to determine whether to list the Program Section definition as

```
        CSECT    0
or      CSECT    1
```

and whether to place 0 or 1 in PROTP (Protection Type for Program Section): STARIBO will use this value in generating the CSECT declaration in object code. NMPTRIG is also tested in SUMMARY, to decide whether to include the string "(NO MEMORY PROTECTION)" after the program size value.

NS No Summary)

At the beginning of Pass 1, if NSTRIG is FALSE, we do a page eject. Note the connection between LS and NS in this regard: normally we do a page eject at the beginning of Pass 1 in order to set off any listings to be produced (Source, Object, Summary); but if LS is not given, and NS is, as would be the case if no listings were desired, then we do not do a page eject in Pass 1. NSTRIG is also sensed in Pass 1 by PRIFSUM (Print if Summary): if FALSE, we force the printing of first and last lines and all entry points. Ordinarily, we would always print these lines, even if no Source Listing is requested; but, if the user does not even want a Summary, we assume he wants to eliminate all extraneous listing. The only other test of NSTRIG is in SUMMARY itself: if the trigger is set, SUMMARY simply exits without doing anything.

PS Partial Summary)

There is a peculiarity in the manner in which PSTRIG is set. Ordinarily, PSTRIG is set if and only if the PS option was given on the control card. However, if the compilation is being done on-line in UTS, and if no options were given, then we set PSTRIG. Thus the default options for an on-line compilation are PS, GO. The only test of PSTRIG is in SUMMARY: if the trigger is set, we skip the generation of the various tables and lists, and go directly to OFS (Output Final Statistics).

RT Real Time

Only one reentrant subprogram can be generated per compilation, so if RTTRIG is set STASCAN gives an error message on any SUBROUTINE, FUNCTION, or Statement Function Definition statement which (1) is not the first meaningful statement in the compilation (i.e., ANYSTAS is TRUE), and (2) the subprogram is not local (i.e., its EXTERNAL trait is set). Pass 2 tests RTTRIG in two places: (1) in Sprog Head Gen (SPHEADGN), to set the Real Time Sprog Trigger (RTSPGTRG) under the appropriate conditions (see 3.78.3), and (2) in the BLC POP, to set the Clobber Bits for all DO-regs, rather than rely on the Clobber Bits associated with the particular library subroutine being BLCed.

S Symbolic In-Line Machine Code)

In Pass 1, STRIG is tested by Label Field Scan (LBLFLDSC in PASS1F) when it encounters a source line with 'S' in column one. If STRIG is set, SINCOLM1 is called to handle the scanning (see 3.57.1). The only other test of STRIG is in PASS3F which uses it to determine whether to list SYSTEM SIG5F or SYSTEM SIG7FDP on the Object Listing.

SB Sequence Binary)

The SB option has the form

SB(name)

where name consits of up to four EBCDIC characters. This name is left in SBNAME. There is also an SBTRIG to indicate the presenece of the option. SBTRIG is tested in OUTBO in its initialization routine STARTBO. If SBTRIG is set, STARTBO will use the contents of SBNAME for the Binary Object record identifier. If SBTRIG is not set, it will use "MAIN" or the first four characters of the first subprogram name.

SBIT Sign Bit

SBITTRIG is tested by Pass 2 in the following subroutines:

GTCCXOWN guarantee sign of value in Condition Codes; own mode; is sign correct? (returns an answer)

GTADXOWN guarantee addressable; own mode; is sign correct? (returns answer)

QUOTGEN quotient gen

The first two subroutines are involved in the handling of relational operators. In all three subroutines, if the operands are integer, and SBITTRIG is set, we require the evaluation of all operands with correct sign, rather than allowing signs to be pulled out.

SI Source Input)

This option tells the compiler nothing. It is allowed for the sake of compatibility with other processors. Though there is a space for the corresponding trigger in a table of option triggers, there is no name attached to it: the name SITRIG does not exist.

SO Source Output)

This option is sensed by Next Input Character (NXTINPCH) in PASS1ML when it reads a new source card. If SOTRIG is set, NXTINPCH copies it out to the SO device by calling SOWRITE.

X Conditional Compilation)

During Pass 1, XTRIG is tested by Label Field Scan (LBLFLDSC) to decide whether to compile a source card with 'X' in column one; it is also tested by Next Active Character (NXTINPCH in PASS1ML) in checking for continuation lines. In SUMMARY it is tested to see whether to list "number of X-cards skipped/compiled".

3.14 Line Numbers

There is some confusion about line numbers, most of it introduced by modifications for XOS. There are now two totally different techniques for line numbering going on at the same time in the compiler, and they are not totally compatible.

Originally, a line number was an actual count of the input cards (or lines), beginning at one and increasing in steps of 1. Its value is kept in LINECNT, and updated by Release Input (RELINPUT), which is called by Release Card (RELCARD) in Pass 1. It is used to determine whether a line is the actual first line in the program, since this line gets printed even when LS is not

requested. Also, in run-time DEBUG mode, this count is sent to Pass 2 as an "interface statement", consisting of two items:

1. LINENRNO

2. Line Count

Pass 2 uses this count in two places. One is for run-time DEBUG check-ins (see Statement Check-in Gen) and the other is in the case of an inexplicable error (probable hardware failure) in Pass 2 (see Pass 2 Abort Exit). If such an error occurs, Pass 2 checks to see whether line numbers are being passed, and if so, it can print out not only the place in Pass 2 where the error occurred, but also the source line number of the statement that caused the error.

In the processing of EQUIVALENCE statements (as well as some other places, such as RETURN and DO), it is useful to specifically include the line number with the other items created for the statement, in order that diagnostics generated later on can refer to the offending statement by line number. Since EQUIVALENCEs are often long and compliacted, occupying several continuation cards, we thought it would be better to give the line number of the exact continuation card on which the error appeared, rather than use LINECNT, which is not updated until the whole statement is released, and therefore never reflects the line numbers of continuation cards. To do this, we created a subroutine, Derive Line Count ML, which computed the current exact line number by dividing the Input Character Count by 80.

With the advent of XOS compatibility, however, this subroutine produces a value that is not just a simple line count. It is the XOS line "label", so to speak, and can have up to three digits after the decimal point (e.g., 1, 1.1, 1.23, etc.) This "label" is created by XOS and included in the source image. Therefore, Derive Line Count ML gets it by determining the actual line count to locate the current souce image in the Input List, then pulling out the XOS "line number" word. Most of the places in the compiler that reference line numbers have now been changed to use the new XOS line numbers <u>if they are present</u>, but the old ones in other environments.

However, not all of them have been changed. The ones that used to use LINECNT still do. This is fine in Pass 1. The two places in Pass 2 would probably be a little strange when running under XOS. The Pass 2 Abort, which should not happen very often (it requires a hardware failure of just the right kind, or a bug in the compiler) can print out a line number that is not reflected on any of the user's listings. And run-time DEBUG will <u>not</u> work under XOS, because it is heavily dependent on the old style of line numbers.

One other comment on line numbers. In the source and object listings, it was specifically designed that the line number would be printed followed by a colon followed by <u>one space</u>, followed by the source image. Amidst great industry-wide inconsistency about the spacing between line numbers and source images (which is more important than it sounds, because it is useful to be able to tell what column your source characters are in), we hoped for a

universal procedure of one space. Unfortunately, this convention applies only to 9300 FORTRAN, the 9300 Mag Tape Editor, and Sigma 5/7 FORTRAN IV. Now, however, this has been changed, and the source images float free, such that it is hard to decide what column things are in. Also, to aid this, the FORTRAN source lines were designed to line up, column for column, with the generated assembly language "source" lines in the object listing, and this is no longer true.

3.15 Plex Passing

One of theproblems of Pass 1/Pass interface is making sure the Plex List looks the same to Pass 2 as it did to Pass 1 at corresponding points. (Nontrivial expressions are represented by pointers to plexes, which may in turn point to other plexes). A simple-minded way of doing this would be to pile up plexes throughout the program, never throwing them away. Then, during Pass 2, the Plex List would be left unchanged and all expression pointers would be valid. Unfortunately, this method is very wasteful of space--most plexes are used only once and never needed again. To save this space, the program is segmented during Pass 1, and only the plexes for one segment at a time are accumulated on the Plex List. Pass 1 sends the plexes (all at once or in installments) to Pass 2 with special interface items: "Load Plex List" and "Accumulate new Plexes".

Outside of DO or REPEAT loops, the program is segmented at statement boundaries, since there is no inter-statement expression remembering going on. Within loops, however, we are attempting to recognize the re-use of loop-independent subscript expressions anywhere in the loop (see 3.3); we therefore would like to retain the loop's plexes for as long as possible. But, still, some kind of segmentation is needed within a loop, so we do the thing that works best for the innermost loop of a nest, at some cost to outer loops: we break off plexes whenever the loop nesting level changes (either upwards or downwards).

The making of such segmentation decisions, as well as the actual plex passing, is done by the subourinte FILECODE in PASS1F. This subroutine is called at the end of every statement during Pass 1; ostensibly, its function is to move Pass 1/Pass 2 interface code from the Code List ot the Output List, but before doing so it deals with the plex interface problem as follows:

1. Pass plexes for the statement just finished. How this is to be done depends on how big the Plex List is now (determined by CNTing PLEXLIST) and how big it was the last time FILECODE was called (remembered in the cell PREVPXCT - Previous Plex Count):

   - If the Plex List is now empty, or its size has not changed from last time, we do nothing.

   - If the Plex List was empty and now has n (n 0) words, we output an interface item that says to Pass 2, "Empty the Plex List and then load these n words into it" (handled by LDPLXLST in PASS2F).

- If the Plex List has grown from m (m 0) to n (n m) words, we output an interface item that says, "keep the m words you and load these n-m additional words"
(handled by ACPLXLST - Accumulate Plex List - in PASS2F).

2. Decide whether to discard the current plexes. This is done whenever we are not in a loop (DOLIST is empty), or the loop nesting level (determined by CNTing DOLIST) is different from what it was last time FILECODE was called (as remembered in PREVDOCT - Previous DO Count).

FILECODE maintains the Plex size and nesting level in PREVPXCT and PREVDOCT for use the next time it is called.

There is one other way in which pelxes are sometimes conveyed from Pass 1 to Pass 2: on the Group List. This only happens in the case of array dummies. When setting up the prologue code for a SUBROUTINE statement, Pass 2 needs to have Plex List set up the way it was during the DIMENSION statement in which the array declaration appeared during Pass 1; this is necessary in order for Pass 2 to be able to properly evaluate the expressions for the array offset, size, and subscript multipliers--these expressions are non-trivial for adjustably dimensioned dummy arrays. See also section 3.78.2.

So far, only the Pass 1/Pass 2 interface has been mentioned in connection with plex passing. This is because Pass 3 is unable to evaluate all but one kind of expression, and therefore usually has no use for plexes. The one exception is the Address Function Plex (ADRFPLEX) generated by SINCOLM1 in response to inOline machine code statements containing the WA, HA, BA, or DA addressing functions. (See section 3.57.) When Pass 2 encounters an In-Line Machine Code statement, it unconditionally passes the current plexes to Pass 3 by generating a "Load Plex List" item.

3.16 Labels

Some labels are statement numbers that appear in the user's source program. This includes local and non-local (global) labels. Others are created by the compiler when it needs to reference part of the program (usually by branching there) without a specific source program request. (For example, logical IF testing often involves branching around to test various parts of the expressions.) These are called created (or generated) labels.

Each of the three kinds is kept on a different list, as shown below:

LOCLBLST Local Label List

NLCLBLST Non-local Label List (statement numbers followed by $)

CRELBLST Created (or Generated) Label List (number followed by G)

These lists are very similar in make-up. Each has a two-word entry, the first word containing the number of the label, and the second word containing the program location at which the label is defined.

Pass 1 initiates the definition of all statement number labels. When Label Field Scan (LBLFLDSC, Section 2.9.5) sees a number in columns 1-5 of a source line, it sends a LABELNO interface item to Pass 2. When this item comes along to Pass 2, the previous statements will have been generated, so the program location will be known and can be filled in.

Created labels are almost always initiated by Pass 2, although Pass 1 does it occasionally (e.g., see Section 3.65.2.1). Again, Pass 2 always fills in the definition location. Furthermore, as an aesthetic improvement, Pass 2 also fills in the label's number as it defines them. This is so that the created labels will come out in numeric order by definition. The labels appear on the CRELBLST in the order they were though up, either as references or definitions. Their numbers are in the order they are defined. This makes them easier to find on the object listing.

Statement numbers (local or non-local) may be classified as "active" labels. This is one of the traits indicated in the upper twelve bits of the first word in the label list entry. It means that the label has appeared in some context where it can potentially be branched to. (This means that it has appeared in a GO TO, ASSIGN, or IF statement, or has appeared in the argument list of a subprogram call.) This is important in generating code that remembers values across statement boundaries. If a label can be branched to, it is not safe to remember values computed in the previous statements. Typically most labels are active, but sometimes the user writes labels on some statements (or all statements) without ever referencing them. This feature helps us generate slightly better code in those cases.

The most important sources of inactive labels are FORMAT labels and DO labels. A DO label is the label mentioned in a DO statement - the label of the last statement in the range of the DO. This label is usually not branched to and is thus an unavoidable inactive label; it is the primary reason for having the active/inactive trait.

Created labels also have traits. Some are active, and some are inactive. Further information on labels can be found in Section 1.5, under the three lists named above.

3.17 Loops

There are four kinds of loops the compiler is concerned with:

- DO loops

- REPEAT FOR loops
- REPEAT WHILE loops
- implied DO loops in I/O lists

Each of these has its own Pass 1/Pass 2 interface item, but they all share a common end-of-loop item. Pass 2 treats implied-DO loops in I/O lists exactly like REPEAT FOR loops, since no-times loops are allowed and the loop step and termination values are explicitly allowed to change during the loop. The only loops for which any meaningful code optimization is attempted are DO loops; see 3.38.2 for details of DO loop code generation.

3.17.1 Properly Nested Loops

During Pass 1 we keep track of loop nesting by means of information on the DO List. Pass 1 uses one three-word item for each curre nt level of looping; these words contain (1) a pointer to the end-of-loop label, (2) the line number of the DO/REPEAT statement (for diagnostic purposes), and (3) a pointer to the control variable, in the case of DO-loops (zero for other loops). The latter is used in determining whether a given subscript expression is "loop-independent" (i.e., depends only on constants and current DO-loop control variables). End-of-loop is sensed by Pass 1 when it discovers that the statement it has just finished had a label, and that that label appears as one of the loop-end labels on the DO List. If it is the bottom-most label, then the loop has been properly nested; we simply generate an End-of-Loop item and discard the three-word group from the DO List. If the statement label appeared on the DO List in some other position, then the corresponding loop was incorrectly nested and is treated as in 3.17.2.

3.17.2 Illegally Nested Loops

A case of illegal nesting occurs when a statement label appears that ends a loop that is not the most recently opened loop. Pass 1 detects this by finding the label on the DOLIST, but not at the bottom of that list. Pass 1's recovery is to print an error warning and remove the loop information from the middle of the DOLIST. If it did not remove the information, the loop would appear to be unclosed at the end of compilation.

In the process of removing the loop information, Pass 1 counts the number of loops that are in the way of the one sought. This number turns out to be useful for Pass 2's error recovery. Rather than a DOLIST to worry about, Pass 2 has the Delayed Code List (DCODLIST) which has a file of code for the bottom of every loop now open. Normally the bottom file of the DCODLIST would be the one Pass 2 needs to put out, because it corresponds to the most recently opened loop. In the case of illegal nesting, however, some other file is needed. The count from Pass 1 tells how many files to move out of the way before putting out a file for this loop bottom. After doing this, Pass 2 puts back the remaining files.

The same procedure is applied to the DOLIST, so the correct information will be on the bottom of that list. The DOLIST contains information about DO loops, their induction variables, and the index registers associated with them. (See 2.17.2).

The recovery just described causes the program to be generated as the user wrote it. It may not be what the user meant to write, but it is executable and could conceivably do what the user wanted if, say, he did not know about the nesting rule. This is a case where the "correct" error recovery seemed about as easy to do as any other, so we did it.

3.18 Source Lines

When an object listing has been requested (LO option), we want to intersperse the source lines at the appropriate places in the generated code. This means sending these lines along (as text strings) as part of the intermediate output between passes. In the compiler's comments, these are often referred to as "compressed" source lines, but in fact they are not compressed at all. The terminology is a holdover from the 9300 compiler, where they were compressed, with blanks counted rather than included in the text. On Sigma we decided that the situations we wanted to optimize were not those in which an object listing was being produced; these will be slow anyway, and it was not worth the space in Pass 1 to go about compressing them.

Output Print Buffer (PBOUT, in PASS1F) moves the entire print buffer onto the Output List. For this purpose, the print buffer (located in FORTTEMP) is preceded by a word count, so that the LOD POP can be used to load the entire "print packet" onto the Output List.

Pass 2 (in SOURCGEN) merely transfers the source lines from its Input List to its Output List, preceded by the P23 indicator that will identify them to Pass 3.

Pass 3 (in OUTSOURC, Section 2.29.3.5) receives the source lines and prints them on the object listing. It is the obligation of Passes 1 and 2 to make sure that the source lines come at the right place, ahead of the coding generated for the statement, and behind the coding generated for the previous statement. As it happens, this takes no special effort; it just falls out.

3.19 Run-Time Debug

The FORTRAN IV compiler generates code to interface with the FORTRAN Debug Package (FDP) when the DEBUG option has been specified. When the DEBUG optrion trigger (DEBUGTRG) is set, additional code is generated for most statements (see 3.19.1); this code calls subroutines in the run-time library associated with FDP. FDP is thus informed of the beginning of new statements, changes in program flow, or changes in the values of variables. The compiler

also accumulates further information for FDP's use: tables which establish the correspondence of source line numbers, statement locations, label numbers, and entry point names (see 3.19.2).

Most of these functions are performed by Pass 2 while generating code for the program. However, Pass 1 has two additional jobs when DEBUGTRG is set: first, it must pass source line numbers to Pass 2; second, it must tell Pass 2 when an executable statement is to follow. The latter is done retroactively by STASCAN in Pass 1 as follows: at the beginning of scanning a source statement, STASCAN files a "Null Statement" indicator on the Code List, and saves a pointer to the location thereof; if the statement was executable, its scan routine exits to one of the Executable Statement Exits in STASCAN, at which point the "null statement" indicator is replaced by an "executable statement" indicator.

3.19.1 Debug Interface Code

There are twelve subroutines in the FDP library which the compiler generates calls upon:

Name	Genned by	Purpose
9DBINIT	(Pass 2 initialization)	
9DBENTER	DBSPHDGN	Subprogram entry point check-in
9DBSCKIN	STCKINGN	Executable statement check-in
9DBACKIN	ASSIGNG	ASSIGN statement check-in
9DBCCKIN	STANSPCG	Standard subprogram call check-in
9DBFHGO	GOTOSTAT	GO TO statement check-in
9DBFHGO	COMPGOSG	Computed GO TO statement check-in
9DBFHGO	ASNGOSTG	Assigned GO TO statement check-in
9DBFHIF	AIFSTAG	Arithmetic IF statement check-in
9DBFHIF	IFOVFLG	IF OVERFLOW statement check-in
9DBFHIF	IFSWSTG	IF SENSE SWITCH statement check-in
9DBFHIF	IFSLSTG	IF SENSE LIGHT statement check-in
9DBFHIL	LOGIFSTG	Logical IF statement check-in
9DBFHRS	RETURNGN	Standard RETURN statement check-in

9DBFHRJ	ASRETSTG	Asinine RETURN statement check-in
9DBDCKIN	(BIF POP)	Value change (data) check-in
9DBMCKIN	(BIF POP)	Multiple dummy value change check-in

Calls on 9DBDCKIN and 9DBMCKIN are generated by the BIF POP when DEBUGTRG is set and the instruction being BIFed is a memory-modifying instruction (its M-bit is set: bit 31).

Most of the statement check-ins pass supplemental information; for instance, the statement GO TO K gets genned as:

```
L       LW,7          K
        BAL,6         9DBFHGO
        GEN,1,14,17   0,line-number,label-number
        BAL,6         9ASGOTO
```

Information to be passed in this way is genned by filing Debug GEN Indicator (DBGENIND) on the Code list, with the 1-bit value in its indirect address bit, the 14-bit value in the op code, register, and index fields, and the 17-bit value represented by a pointer in the next word. The 14-bit quantity is often a source line number; in the Label Table (see 3.19.2), it is the number (in Entry List) of the subprogram entry point associated with the label. The 17-bit quantity is often a label number: the number of a local label, or 100000 plus the number of a global label. The subroutines GETLNLBL (Get Line/Label Words) and GENLNLBL (Generate Line/Label Words) build and generate the item:

```
        GEN,1,14,17   0,line-number,label-number
```

Calls on 9DBSCKIN are generated by Statement Check-in Gen (STCKINGN) in response to an "Executable Statement" Indicator. Other FDP calls are genned by the appropriate statement gen routines in PASS2F.

3.19.2 Debug Tables

There are three tables accumulated during Pass 2 for the purposes of the debug interface:

1. Source Line Number Table: its one-word entries are of the form

```
        GEN,1,14,17       t,sln,loc
```

where t=1 if and only if the source line has a label (and there is an entry for it on the Label Table); sln is the source line number; and loc is the location of the statement code.

2. Label Table: its one-word entries are of the form

```
GEN,1,14,17          d,ent,lbl
```

where d=1 if and only if this is the label of a DO/REPEAT range; ent is the entry point identifier: 0 for main program labels, n for labels belonging to the entry point corresponding to the nth Entry Table item; lbl is the label number (plus 100000 for a global label).

3. Entry Table: its three-word entries have the form

```
GEN,1,14,17          e,0,loc

TEXT              'name....'
```

where e=1 if this is an ENTRY name, e=0 for a SUBROUTINE, FUNCTION, or Statement Function name: name is the eight-character name of the entity.

These tables are built up on three lists in the form of Pass 2/Pass 3 interface code, like instructions. Each of these lists therefore functions like the Delayed Code List. At the end of the program, these lists are copied (by Debug Table Gen, DBTBLGEN) to the Code List and run through the assembler (ASMCODE). The lists are:

```
SRCLNLST  Source Line Number List

LABELLST  Label List

ENTRYLST  Entry List
```

SRCLNLST and LABELLST are updated by the subroutine Register Source Line/Label (REGSRCLN) which places one item on SRCLNLST and, if the current statement has a label, one item on LABELLST. The SRCLNLST item consists of a Debug GEN Indicator including the curret line number (from LINENUMB) and label presence bit, followed by a label pointer representing the statement code location. This is the current statement label (CURRLABL) , if present; if there is no label on this statement (i.e., if CURRLABL contains zero), we make and define a label (MDEFLBL) for use in SRCLNLST. Now if CURRLABL is nonzero, we make a LABELLST entry for it. This consists of a Debug GEN Indicator containing the DO-label bit and entry point number, followed by the label number. The DO-label bit is determined by SOL DOLABEL; the entry point number is obtained by CNTing the ENTRYLST; and we call GETLBLNO to get the label number, to which we append ABSINDIC (Absolute Address Indicator) so that this word looks like an address pointer.

REGSRCLN is called from the following places: FORMATSG (FORMAT statement gen), CODESTAG (In-line Machine Code statement gen), STCKINGN (Executable Statement Check-in gen), and ASFDSTAG (Arithmetic Statement Function Definition gen).

ENTRYLST items are built by REGSPROG (called by SUBROUTINE Statement gen and FUNCTION Statement gen) and REGENTRY (called by ENTRY Statement gen). Each five-word item is composed of (1) the Debug GEN Indicator containing the ENTRY bit, (2) the subprogram pointer, (3) a Hollerith two-word Absolute Indicator (HABS2IND), and (4,5) the two words of the subprogram's name obtained from the Symbol Table.

3.20 ABNORMAL Statement ABNORMST _ Pass 1 only)

This statement no longer does anything, and is present only to avoid syntax errors. Each identifier in the list is given the ABNORMAL trait, but this trait is defined (in LISTDEFS) as zero, and so has no effect whatsoever. Nonetheless, all the mechanism is present in the plexes and inheriting to handle the trait, should it ever be needed.

Originally, it was thought that "ABNORMAL" standing alone, with no identifiers, would mean that nothing was abnormal (whereas the absence of any ABNORMAL statements would mean that everything was abnormal, so that unsuspecting users would not be hurt by optimization), but since the compiler does not do anything that requires knowledge of the abnormal trait, this form is retained only for possible compatibility. To have any effect, a trigger would have to be set just before exiting.

Exit is made through Declarative Exit 1 (indicating the statement is invalid in a BLOCK DATA subprogram).

Nothing is transmitted to Pass 2 and no change to the Symbol Table occurs.

3.21 ACCEPT TAPE Statement ACCTAPST _ Pass 1 only

After finding the letters "ACCEPT", set the Fail Exit to return to the ACCEPT statement if ACCEPT TAPE fails (see TRY-Fail, 2.7). Then loop for "TAPE". If not found, fail immediately. Otherwise proceed, but remember that a major error later on in the statement can still cause it to fail and go try ACCEPT. For example, the statement

```
ACCEPT TAPE, LIST
```

is a legal ACCEPT statement, but fails as an ACCEPT TAPE statement.

File the ACCTAPNO (same as RDBCDNG), then go to FORTRAN II Input Scan with WO containing a pointer to the paper tape input unit number (103).

3.22 ACCEPT Statement ACCEPTST _ Pass 1 only

Control comes here only if ACCEPT TAPE fails (usually because the letters "TAPE" were not found). Since things have been put back the way they were at the beginning of Statement Scan, we have to rescan the wrod "ACCEPT" (which we know is there). Then file the ACCEPTNO (same as RDBCNO) and go to FORTRAN II Input Scan with WO containing a pointer to the typewriter input unit number (101).

3.23 ASSIGN Statement

3.23.1 ASSIGN Statement ASSIGNST _ Pass 1)

File ASSIGNNO followed by a pointer to a statement number (on either the Local Label List or the Non-local Label List). See Labels, Section 3.16, for a description of what it means to be an active statement number. Then file a pointer to a variable of any mode. Exit through Active Exit.

Three words are transmitted to Pass 2:

1. ASSIGNNO
2. Statement Number (pointer to one of the label lists)
3. Assigned variable (pointer to Symbol Table or plex, if subscripted)

3.23.2 ASSIGN Statement Gen ASSIGNG _ Pass 2

The ASSIGN Statement indicator is followed by two additional input words: a pointer to the label being ASSIGNed, and a pointer to the variable it is to be ASSIGNed to. The code generated for the statement

```
ASSIGN 23 TO V
```

is as follows:

```
        LW,9    1G
        STW,9   V
        . . .
1G      B       23S
```

The in-line code (LW and STW) are generated by ASSIGNGN; the "label constant" B 23S is generated at the end of the compilation by OLBLCON (Output Label Constants). Having obtained the label pointer (23S) from the input list, ASSIGNGN calls REGLBCON (Register Label Constant) to obtain the corresponding Created Label pointer (1G). REGLBCON searches the Label Constant List (LCONSLST) for the label; if found, it returns the associated Created Label pointer; if not found, it creates a new Created Label and forms a new LCONSLST entry.

If the Debug option (DB) had been specified, the code described above is preceded by the debug ASSIGN Check-in:

```
        BAL,6     9DBACKIN
        PZE       n
```

where n is the label number (e.g., n=23 in the above example) for a local label; n is 100000 plus the number of a global label. This number is derived by the subroutine GETLGLNO (Get Label Number) from information in the appropriate registered label list item.

3.24 Assigned GO TO Statement

3.24.1 Assigned GO TO Statement GOTOSTA _ Pass 1)

"GOTO" has already been found (see GO TO Statement, 3.55.1). Call Any Mode Variable Scan and get a variable (possibly with subscripts). If there are statement numbers enclosed in parentheses following this (as allowed for compatibility with old systems), scan them and throw them away (except that, of course, they get registered on a label list and therefore had better be defined somewhere). It is not necessary to indicate that they are active labels; this will be taken care of if they ever appear in an ASSIGN statement. File ASNGONO and the variable. Exit through Jump Exit.

Two items are transmitted to Pass 2:

1. ASNGONO
2. Variable (pointer to Symbol Table or to one kind of Subscripted Array Plex)

3.24.2 Assigned GO TO Statement Gen ASNGOSTG _ Pass 2

The assigned GO TO statement is handled through a library call upon 9ASGOTO. We calculate the address of the variable we are going to in index 7, then BAL,6 to 9ASGOTO. The simple case amounts to just a Load Immediate instruction to register 7; more complicated cases can involve subscripting and/or indirect addressing.

Should the DEBUG option be specified, we precede the BAL,6 by a check-in call upon 9DBFHGO.

3.25 Assignment Statement

3.25.1 Assignment Statement ASIGNMNT Pass 1

Control comes here only after the "quote" statements have failed. The character counts and other data have been restored the way they were at the beginning of Statement Scan. We call Any Mode Variable Scan to get a variable pointer. Note that a variable may be a simple scalar, an array element, or a multiple dummy element. Replacement Scan is used to get the rest of the statement, which may consist of several replacements (or assignments). For example:

```
X = Y = A(3) = 3*Z + 4.6
```

The X and = are scanned here, and everything else is handled by Replacement Scan. The latter is also, of course, willing to find only a single expression following the equal sign, as in:

```
X = Y          or          X = Y + SQRT(Z)/ALPHA
```

We file the ASNMNTNO, the variable, and the replacement (which may be any of all sorts of pointers -- a replacement plex, any type of expression plex, a subscripted array plex, a pointer to a constant list or the Symbol Table, etc.). Check for, and mark, any excess right parentheses. Then exit through Active Exit.

Three items are transmitted to Pass 2:

1. ASNMNTNO
2. Variable to left of equal sign
3. Expression or replacement to right of equal sign

3.25.2 Assignment Statement Gen ASNSTAG _ Pass 2)

An assignment statement is represented by a Replacement Plex, the pointer to which is the next input item after the Assignment Statement indicator. ASNSTAG obtains this pointer and calls the subroutine ASNMNTG (Assignment Gen) which, in turn, calls GTADUOWN (Guarantee Addressable by its Own Mode) to gen the code to evaluate the "right hand side" and store into variable(s). See the description of Replacement Gen (REPACGEN) in section 3.10.2.7 for details of this process.

3.26 BACKSPACE Statement

3.26.1 BACKSPACE Statement BKSPACST _ Pass 1

There may any number of units specified, separated by commas. Each unit may be represented by an expression whose mode can be converted to integer. For each unit, file the BKSPACNO followed by a pointer to the expression. Exit through Active Exit.

If more than one unit is specified, they will be transmitted to Pass 2 as separate BACKSPACE statements. Each one contains two items:

1. BKSPACNO
2. Unit number (pointer to expression)

3.26.2 BACKSPACE Statement Gen BACKSPSG _ Pass 2)

The BACKSPACE statement is very easy to gen (see also REWIND). In addition to the statement indicator, Pass 1 sends an expression pointer for the unit number to be backspaced. With this pointer in WO we simply JRS to Grntee Integer In Accumulator (GTACUINT), then build a library link to 9BKSPACE.

In fact, BACKSPACE Statement Gen shares code with several other easy statement generators, so it starts out by ADRing the 9BKSPACE Link, then later generates the link through PO.

3.27 BLOCK DATA Statement BLKDATST _ Pass 1)

This statement has no effect and is accepted only for compatibility. It does, however, set a trigger (BLDATRIG) that determines two things: only declarative and DATA statements will be allowed in the program; the resultant object module will not be a main program. It will be a subprogram without a name.

A BLOCK DATA statement cannot be under a Logical IF and must be the first statement in the program; otherwise a diagnostic is given. This is an important line, which must be printed (even if no LS) as long as there is at least a summary being printed (see PRIFSUM). Exit through the DATA Statement Exit, which does not have to check for previous BLOCK DATA (not possible if this is the first statement) or Logical IF (already checked).

Transmits nothing to Pass 2 (except the status of the BLOCK DATA Trigger).

3.28 CALL Statement

3.28.1 CALL Statement CALLSTA _ Pass 1

File CALLNO. Use Call Statement 1 (CALLSTA1, below) to process the call and file a function plex. If there are additional subprogram names, separated by commas, each of them is processed as described above. This is an undocumented feature carried over from the 9300 compiler, and should probably be documented. Exit through Active Exit.

Two items are transmitted to Pass 2:

1. CALLNO
2. Function plex pointer

3.28.2 CALL Statement 1 CALLSTA1)

Used by CALL statement and CONNECT statement. Gets subprogram name and checks permissibility of CALLED. If illegal, it gives a diagnostic and jumps down to process the arguments anyway, in order to pick up possible diagnostics there.

Otherwise, it registers the SPROG trait, which establishes the calls for this identifier, and sets up the permissibility of CALLED (See Permissiblility, 2.8.1). Then call Argument Sequence 1 (ARGSEQ1, an alternate entry point to ARGSEQSC, Section 3.10.1.3.5.1) to scan for arguments and build an Argument Plex (which may be empty). Now build a Function Plex from the subprogram name pointer and the Arg Plex, and file this.

3.28.3 CALL Statement Gen CALLSTG _ Pass 2)

There is not much to this statement generator. Only four POPs. This is because all the work is done in a subroutine called Sprog Call Gen (SPCALLGN). At the statement level we JRS to NXINWORD to get the Function Call Plex pointer. (This plex works for FUNCTION and SUBROUTINE calls.) Then we JRS to SPCALLGN, which returns with a pointer to the name of the subprogram in W0. We do not need this, so we throw it away and BRU to Statement Exit (STATEXIT).

3.28.4 Sprog Call Gen SPCALLGN)

First we pull the Function Call Plex, which yields two things: W1 is the Sprog Name Pointer, and W0 is a Sprog Arg Group Plex pointer. We check the sprog name pointer to see if it is Intrinsic. If so, we fall into a little routine called Intrinsic Sprog Call Gen (INTRINSG), which filters through a large branch table to the appropriate intrinsic function generator. There are no intrinsic SUBROUTINEs (just FUNCTIONs), but the generator would work just fine if there were.

It is Standard Sprog Call Gen (STANSPCG) which generates standard calling sequences. There are three parts to a standard call: the argument setup part in which arguments and argument addresses are computed; the actual call (consisting of a LI,14 and a BAL,15); and the calling sequence, with one word per argument. The first and third parts are actually generated simultaneously, with the code for the third part being delayed on the Delayed Code List (DCODLIST).

We start by reserving the DCODLIST and pulling the Sprog Arg Group Plex to the ARGLIST. Then we count the ARGLIST and retain this count for awhile in WO. Now begins the loop. We take an argument off the top of the ARGLIST. (If there are none left, we jump to the bottom of the loop.) We guarantee it addressable with GTADUOWN (Grntee Addressable Own Mode) in case it is an expression. This routine is deceptively named, for it does more than just expression evaluation. It is actually designed to be called to get a subprogram argument ready to use, for it checks for such things as subprogram names and labels, which cannot appear in expressions but can appear as subprogram arguments.

If the argument is a label, we register it as such on the Label Constant List. This registering is done by Register Label Constant (REGLBCON), which is described under the ASSIGN statement, section 3.23.2. If it is a subprogram name, we call Update Pointer ML (BRL to UPOINTML) just in case the thing has acquired mode later on. Then, if it is neither a DUMMY nor a SPROGDEF, we register the trait EXTREF, indicating that this must be externally referenced. The fact that all this is done in a routine (GTADUOWN) whose name does not hint that it exists only for subprogram arguments is rather surprising.

But to continue with the discussion of arguments, we derive the op code to be used in the calling sequence. This is done in a subroutine called Derive Calling Sequence Op Code (DCALSEQ) that uses the mode field to govern the op code. If the pointer in WO is a constant or a temp, it appends a protected bit (P) to the R field of the calling sequence op code. This is stored in OPTEMP to be BIFed or BINed upon later.

There is still some addressing logic to go through. We check to see if the argument is a non-trivally subscripted array. This means we check to see if it is

a) a Fully Subscripted Array Plex, or

b) a Constant Subscripted Array Plex whose array name is a DUMMY.

In either of these cases, we must compute an address and store it in a temp, then reference the temp indirectly in the calling sequence. The address computation is done in Index Register 7.

After generating any setting-up code for the argument, we reserve the CODELIST and generate the calling sequence word by BIF or BIN OPTEMP. Then we copy the

code just generated over to the DCODLIST since it has to come out later. We are now at the end of the loop that is done once for each argument.

Coming out of the loop, we check the DEBUGTRG. If it is on, we generate a call on 9DBCCKIN (Debug CALL Check-In). Now by careful planning, the argument count, which was computed with a CNT quite some time ago, is in W0 ready for us to BAF a LI,14 instruction. Now, just before BIFing a BAL,15, we check to see whether to REGister the subprogram as EXTREF. As was done earlier for subprogram names used as arguments, we SOF for SPROGDEF or DUMMY and, if neither, we register it as EXTREF, the ZAP the SYTCHLST to get rid of the evidence. Also just before generating the BAL,15 we FET W0 to create an extra copy of the subprogram name pointer. The CALL statement does not need this, but the function call processor does. We exit through CLOBALL (Clobber All Registers) to forget the contents of all index registers.

3.29 COMMON Statement COMMONST _ Pass 1 only)

If the first specification (or grouping of variables) is blank COMMON (i.e., no slash or two consecutive slashes), signal "blank COMMON" on the Param List. If one slash does appear, get the block name and see if it is already on the Block Name List. If it is not, put it on, followed by four blank words that will be used later by Allocate. NOTE: There are only two kinds of names that are not accumulated in the Symbol Table -- Block Names, because they can be the same as other names in the program, and special identifiers that are externally REFed (e.g., 9SIN and F:108) because they are non-FORTRAN names.

Put a pointer to the block name onto the Labeled Common List and signal "Labeled COMMON" on the Param List.

Now, for each variable found, check the permissibiility of COMMON (blank or labeled as signaled in P1). If illegal, give warning (see IDConflict 1 (IDCON1)), allow dimensions, but do not put in COMMON. If legal, set up permissibility and register the trait of whichever type of COMMON is signaled in P1. Allow dimensions (see Allow Dimensions, 3.37), and put the name pointer on either the Blank COMMON List (BLCOMLST) or the Labeled COMMON List (LBCOMLST).

If another slash is found, start this procedure again. When done, exit to Declarative Exit 2 (valid in BLOCK DATA).

Nothing is transmitted directly to Pass 2 on the Output List, but information is accumulated on other lists.

3.30 COMPLEX Statement COMPLXST _ Pass 1 only)

See Type Statements, Section 3.80.

3.31 Computed GO TO Statement

3.31.1 Computed GO TO Statement GOTOSTA _ Pass 1)

"GOTO(" has already been found (see GO TO Statement, Section 3.54.1). Use Active Label Scan to pick up any number of statement numbers, separated by commas, and pile them on the Work List, which has been reserved. Then build a Label Group Plex from these labels. Allow a comma after the right parenthesis, then go scan an integer expression (which may not actually be Integer, but can be converted to Integer). File COMPGONO, followed by the expression pointer, followed by the Label Group Plex pointer. Release the Work List and exit through Active Exit; not Jump Exit, because IBM's foolish changes have defined it so that you can fall through a computed GO TO if the controlling value is out of range, which ought to be detected as an error.

Three items are transmitted to Pass 2:

1. COMPGONO
2. Control value (pointer to expression)
3. Pointer to Label Group Plex

3.31.2 Computed GO TO Statement Gen COMPGOSG _ Pass 2)

Code generated for the computed GO TO statement has changed slightly since the original version of the compiler. Here are the old and new versions of the same code.

```
GO TO (7,8), K                GO TO (7,8), K
     (old code)                     (new code)

     LW,7  K                       LW,7   K
        BLEZ  2G                        BLEZ   2G
        CI,7  2                    CI,7   2
         BLE   1G,7                  BG        2G
1G       B     2G                    1G  B      1G,7
         B     7S                  B       7S
     B     8S                    B      8S
2G     EQU   $                 2G  EQU    $
```

The two versions of the code are the same length, but in the normal case (K within range) the old version would run 1.9 microseconds faster because the BLE instruction would branch. It turns out the code was changed because of a peculiarity of the Sigma hardware. Suppose K were absurdly large--say several thousand. Then the computed GO TO statement might be tempted to branch to code whose memory protection type would not allow it to be branched to--say another user or part of the monitor. "Aha," you say, "the branch would never

be executed; the testing code would prevent it." This is quite true. However, if the BLE instruction were executed, the legality of the effective address (1G plus the contents of Register 7) would be tested before the instruction decided whether or not to branch. In this case it would fail because of a memory protection trap. Thus it becomes illegal to even think about branching into protected code. Even if the conditional branch does not branch, you get trapped out for even considering the branch.

COMPGOSG calls NXINWORD to get the expression pointer. It wants the value of this expression in index register 7. It checks the pointer for Integer Addressability; if TRUE, it loads it directly into 7; if FALSE, the expression is guaranteed Integer into register 9, then a LW,7 9 instruction is generated. This simplistic approach to the expression generation sometimes causes a redundant instruction to be generated, but the case of an expression being anything other than Integer Addressable (which is all that IBM allows--they require an Integer Scalar) seemed rare enough that extra code in the compiler to make super-nice code at execution time seemed wasteful.

After we have the expression in 7, we check the DEBUGTRG: if TRUE, we generate a check-in call on 9DBFHGO. Then we create a label (2G) and generate the testing code. To generate the CI instruction we have to get the Label Group Plex, PUL it and CNT it to see how many labels there are. Then we make another label (1G), define the label, and generate an indexed branch to it. Now we generate individual B instructions to the labels in the list. Each of these labels is allowed to be an assigned label rather than a statement number. This feature is handled through a subroutine called CHEKLABL that registers assigned labels. If the label is assigned, it is registered on the ASNLBLST and a created label is returned in WD. The generated B instruction is to this created label. Near the end of the statement, we call Generate Assigned Labels (ASLBLGEN) to put out code needed for any assigned labels.

Finally we define the branch-aound label (2G) and BRU to Statement Exit (STATEXIT).

3.32 CONNECT Statement

3.32.1 CONNECT Statement CONECTST _ Pass 1)

File CONECTNO. Use Get Constant A (GETCONA) to scan a constant. Make sure it is either Hollerith or hexadecimal. If not, diagnose as illegal type and proceed anyway. If it is hexadecimal, make sure it is within range (i.e., X'50' through X'13F'). If not, diagnose as out of range and continue. If it is Hollerith, make sure the right-hand two characters are blanks (i.e., it is a 2H constant). If not, diagnose as out of range and proceed anyway.

File the constant pointer. Then use Call Statement 1 (CALLSTA1, under CALL Statement) to scan the subprogram name and arguments, and to produce and file a Function Plex. Exit through Active Exit.

Three items are transmitted to Pass 2:

1. CONECTNO
2. Interrupt location or label (pointer to Hex Constant List or Hollerith Constant List)
3. Subprogram to be connected (pointer to Function Plex)

3.32.2 CONNECT Statement Gen CONECTSG _ Pass 2)

The CONNECT statement generates two pieces of code: the first (which we will call Initialization code) establishes control of the given interrupt, associating with it the location of the second piece of code (the Interrupt code) to be executed whenever the given interrupt occurs. This latter code includes an implicit CALL on the given interrupt routine. The combined code looks like this:

initialization code:

```
                LI,9 int#       interrupt number
                LI,8 1G         interrupt code location
                LI,5 1PTMP      TCB block address
                BAL,6   9CONNECT get interrupt control
                B       2G      branch around interrupt code
```

interrupt code:

```
          1G    BAL,6   9RENTER     initialize flags, pointers, et
                .
                .   (CALL interrupt subroutine)
                .
                .
                .
                BAL,6   9REXIT      restore context and release in err
          2G    EQU     $           resume in-line code sequence
```

The library subroutine 9CONNECT performs the monitor CAL that establishes user control of the given interrupt number; the monitor is told to branch to the interrupt code (1G above) whenever the interrupt occurs. The "TCB block address" (1PTMP above) is the address of a 26-word block needed by RBM to hold certain interrupt-dependent context. The 9CONNECT routine operating under monitors other than RBM ignore the register-5 contents, though the block is allocated in any case. There is one such block for each CONNECT statement. The TCB must begin on an even boundary, so it is allocated by (1) getting a Double Precision Private Temp (via PTEMPMK), then (2) adding 24 to the Private Temp Counter (PTEMPCT).

If the interrupt subroutine is being re-entered (see 3.12), the call on 9RSAVE in its prologue will cause outer ocntext to be saved. The 9REXIT call in the interrupt code sequence will then restore that context upon exit. Whether or not there is context to be restored, 9REXIT finishes by performing the monitor CAL that releases the interrupt and returns to the interrupted location.

3.33 CONTINUE Statement CONTINST _ Pass 1 only)

This routine does nothing, but differs from the Null statement (Section 3.64) in that it is a real statement and can have a label (ought to, in fact). Goes to Active Exit (ACTIVEX).

Transmits nothing to Pass 2 (except probably a label).

3.34 DATA Statement

3.34.1 DATA Statement DATASTA _ Pass 1)

The DATA statement is probably the most complicated in the whole compiler, especially so in Pass 1, since virtually all the work is done there. EQUIVALENCE can get quite complicated, but that happens in ALLOCATE as sort of a whole separate process.

The processing of DATA statements in Pass 1 takes place in two parts. The first occurs while the statement is being scanned. A file of DATA Pairs is created on the Work List. Then, after the whole statement has been scanned, Expand DATA Pair Sequence goes through the DATA Pairs and expands them, matching variables with constants and creating a DATA Item for each match. Passes 2 and 3 then have a very simple construct to work with, consisting of a variable and addend matched with a constant. Much goes into producing that simple construct, however.

The two items that make up a DATA Pair, as prepared for expanding, are a variable list and a constant list, each represented by a plex. For example, the following statement would have two DATA Pairs, each with a variable list and a constant list:

```
DATA X,Y,Z/3,2*3.14159265/ , (A(I), I=1,8)/0/
```

Variable lists can have many different components, including scalars and implied DO-loops as shown above. Constant lists can have individual constants or repeated groups of constants. The complete structure of DATA Pairs is shown below:

I. Variable list.

Each variable list is one or the other of the following:

A. A DATA Variable List Plex (DVARLSPX) -- which combines any selection of the following:

a. A scalar

b. An array name (unsubscripted)

c. Another DATA Variable List Plex (caused by parentheses)

d. A DATA Variable List With DO Plex (parentheses and implied DO loop)--see below

e. A Character Count Plex (used for array elements). It has two parts:

1. The character count of the last letter in the array name. This is used to go back and put a diagnostic on the array name if a subscript is out of range.

2. One or the other of the following subscripted array plexes:

a) A Constant Subscripted Array Plex (CONSUBPX), which has two parts:

1) The array name

2) Offset from zeroth element (a pointer to the Integer Constant List)

b) A Fully Subscripted Array Plex (FULSUBPX), which has three parts:

1) The array name

2) Non-constant part of offset (a pointer to an integer expression on the PLEXLIST)

3) Constant part of offset (a pointer to ICONSLST)

8. A DATA Variable List With DO Plex (DVLWDPX) -- consisting of six items:

1. A DATA Variable List Plex (above)

2. The (Integer) control variable -- This is a pointer to the DSUBLIST, not the Symbol Table. DATA statement control variables are unique entities, and we do not pay too much attention to what they are in the outside world.

3. The character count of equal sign in the DO control. This is used to mark a diagnostic if the DO loop runs an excessive number of times (such as increment of zero).

4. Initial expression (beginning value for DO) -- This and the two following expressions are DATA expressions, which may have the same form as those described for subscripts above. Note that a DO's controlling expressions may reference only outer DOs' control variables.

5. Terminal expression

6. Increment expression

II. Constant list.

Each constant list consists of a DATA Constant List Plex, which is built from any number of items, each of which may be one of the following:

A. A Character Count Plex (used for single constants) -- Like that used for array elements (above), it has two parts:

1. A pointer to a constant, which may be any mode (I,R,D,C,J, or K). A complex constant is treated as two reals unless it is preceded by a plus or minus sign.

2. The character count at the last character of the constant. This is used to mark a diagnostic if the constant is the wrong type to match with the appropriate variable. (This matching is not done until later, in Expand DATA Pair Sequence.)

B. A Repeated Constant Plex, which also has exactly two parts:

1. An integer constant (the DATA multiplier)

2. Either of the following:

a) A Character Count Plex, as in II.A above -- This is used for a single repeated constant (e.g., 3*4)

b) Another DATA Constant List Plex -- This used for repeated sub-groups of constants (e.g., 3*(4,5,6.2)).

C. Another DATA Constant List Plex -- Used for nested sub-groups without multipliers (i.e., redundant parentheses)

The overview of the DATA statement is fairly simple. For each construct of the form:

variable list /constant list/

we call DATA Variable List No DO Scan and DATA Constant List Scan to produce a DATA Pair as two items on the Work List. During the scanning of variable lists, if any non-constant subscripts appear, the variable names are registered on the DSUBLIST (DATA Subscript List), each as a three-word entry:

1. Pointer to name in Symbol Table

2. Character count of last letter of variable name

3. Flag indicating whether this variable has yet been used as a DO control variable in this variable list

This list serves three purposes:

1. These names must all show up as DO control indices.

2. No DO index should appear which has not been used in a subscript (or inner DO control expression).

3. Expand DATA Pair uses these three words as running temps when it actually "executes" the DO loop.

Therefore, whenever a new DO control variable comes along, we can check to make sure that it is on the DSUBLIST, indicating that it has been used somewhere inside. If it has not, that is a not very serious, but nonetheless diagnosed, error. In addition, after scanning each variable and constant list, we can check to make sure that no variable subscript has appeared that is not under control of a DO. This is done by Check DATA Subscripts. Then we have to reserve the DSUBLIST, since this information is no longer pertinent to scanning (we do not want to know about control variables in previous variable lists), but the expander needs those three words for each DO index.

The DATA statement (originating in the 9300 compiler) was not designed in a way that would allow it to fit very well with other features that came along subsequently. In particular, the DATA statement's use of the Plex List causes serious conflicts with compound statements and with handling of DO loops. When the DATA statement was designed, the Plex List was always empty at the beginning of each statement. The DATA statement may create large bundles of plexes (as is obvious from the above outline of what a DATA Pair can look like), and they are only needed at that time; once Expand DATA Pair Sequence is finished, the plexes are no longer needed. We do not particularly want to burden the Pass 1/2 interface file with all these unneeded plexes, so we just threw them away by ZAPing the Plex List. Unfortunately, both compound statements and DO loops now can cause Plex List information to be kept around for more than one statement (See File Code, 2.9.8). So we cannot have DATA statements ZAPing that useful informaion.

Ordinarily it is not recommended to RSV the Plex List. The files thus produced cannot be moved around safely, since plex pointers are relative to the BASE of the Plex List. Conceivably the DATA statement could RSV the Plex List (and then RLS it when through), since it is not planning on doing anything with those plexes except right where they are. However, there are other problems. The DATA statement mechanism is also used by Type statements

(e.g., INTEGER), which IBM decided to allow data to appear in. If the DATA statement reserved and released the Plex List, the Type statements would have to also. But this causes problems because Type statements also allow dimensions to appear, and the dimensions processor (see Allow Dimensions, 3.37) also builds plexes, which it does want to save for later (on the Group List). The dimensions processor will not work well at all if there are Reserve marks on the Plex List; when it copies its plexes to the group list, they will not be offset from the BASE correctly. And unfortunately, the DATA processor cannot reserve the Plex list, do its thing, and release the Plex list all at once. The reserve has to be before scanning, and the release cannot come until after expanding, and in between can come more dimensions, as for example in:

```
INTEGER ALPHA/3.14/ , BETA(N ,M) , GAMMA/4.13/
```

The net result of all this is that we had to disallow DATA (and adjustable dimensions) inside compound statements and DO loops. (See Compound Statement Scan, 2.9.3, and Fail If Inside DO Loop, 2.9.9.4). We also check to make sure that the statement is not going to fail for some other reason (delayed fail, under logical IF), and then call Expand DATA Pair Sequence to pull the DATA Pairs off the Work List and produce DATA Items on the Output List. Then we ZAP the lists we were using, and exit through DATA Exit, which is like Declarative Exit 2 (valid in BLOCK DATA) except that the test for logical IF has already been done.

Information is transmitted to Expand DATA Pair Sequence on the Work List and Plex List, as described above. For each actual variable-constant matchup that results (an array name can produce many), 4 or 5 words are transmitted to Pass 2:

1. DITEMNO
2. Addend value (pointer to Integer Constant List)
3. Origin of variable (pointer to Symbol Table)
4. Constant (pointer to one of the constant lists)
5. Constant for imaginary part of Complex variables

Pass 2 knows whether the 5th word is present by whether the variable pointer (word 3) has mode COMPLEX or KOMPLEX.

3.34.1.1 DATA Variable List No DO Scan DVLNDSCN

This is the outermost level, where implied DO loops cannot appear. To have an implied DO, it must be enclosed in parentheses, and whenever parentheses

appear, we recurse down to DATA Variable List (With DO) Scan. This occurs in DATA Variable Item Scan. At this upper level, all we do is call DATA Variable Item Scan over and over, collecting variable items on the Term List, which we then build into a DATA Variable List Plex and leave on the Work List. This is the first of the two words constituting a DATA Pair. Because DATA Variable Scan is also used by the lower level routine (which allows DO loops), and because we used to allow the CDC form of DATA statements, there is some strange coding in which, if DATA Variable Item Scan finds an equal sign (which it should not), it comes back and tells us, and we turn around and go right back in (through the entry DVARSCN1) to proceed just as if the equal sign were not there. It will eventually be detected as a syntax error.

3.34.1.2 DATA Variable List With DO Scan DVLWDSCN)

This is called whenever a left parenthesis has been found, and is the same as above except that it allows implied DO loops. If it finds one, it builds a DATA Variable List With DO Plex instead of a DATA Variable List Plex.

At each level of nesting, we reserve the Term List so we can play with it, and then release it upon exit. We also reserve the Inner DO List, but unreserve it on exit. This assures that the search for inner DO conflicts will find only inner DOs, not those sideways. (The sideways ones may be hiding behind the reserve marker so that they can be caught when we get to outer levels.) You will have to try this once (I did) to see that it works.

We repeatedly call DATA Variable Item Scan to get individual items (which may turn out to be another whole list, enclosed in parentheses). After the first item, an item may turn out to actually be a DO control variable, if it is followed immediately by an equal sign. In this case, we do not care anything about the name except that it be Integer. Then we search the Inner DO list to make sure this does not conflict with an inner control variable, and we put this on the same list, so that outer DOs will be able to check for conflict with this. Then we search the DATA Subscript List, to make sure that this control variable has been used somewhere inside in a subscript (or the control expressions of an inner DO). Theoretically it is not illegal to have this situation, but it is unlikely to be very meaningful if the control variable is never used.

For each variable appearing as a subscript in a DATA statement, there is a three-word entry on the DATA Subscript List. The third word is a signal of whether the variable has yet been used as a DO control variable. We can now signal that it has.

Then we scan three DATA Expressions (see below), substituting the value 1 if the third expression is absent, and build a DATA Variable List With DO Plex.

3.34.1.3 DATA Variable Item Scan (DVARSCAN)

If the first character encountered is a left parenthesis, we recurse back to DATA Variable List With DO Scan. Otherwise, it must be a scalar or array. If it is either a scalar or an unsubscripted array, we can return with just the pointer to the name. The only other thing we have to do is check to make sure the name is permissible to appear in a DATA statement, and set up new permissibility reflecting the fact that it now has appeared in a DATA statement. It is also necessary to do one other thing. The permissibility of DATA prevents a variable from appearing in blank COMMON, and vice versa. However, a variable can get into blank COMMON through EQUIVALENCE, which will only be detected later on in ALLOCATE, by which time the permissibility word in the the Symbol Table is no longer around. (The word that contains permissibility during Pass 1 is used as an allocation word later.) Since this was (I think) the only place where permissibility was needed after it was gone, we get around this by registering a SOF trait, called DATA, which ALLOCATE can test for.

The only tricky case is a subscripted array. In this case, we need to scan the subscript expressions (using DATA Expression Scan), pile them up on the Script Expression List, and call Combine Subscripts to build some sort of subscripted array plex out of the whole mess. Then we take the subscripted array plex, combine it with the character count at the last character of the array name (for diagnosing out-of-range subscripts), and build a Character Count Plex (not a very descriptive name for what it is really doing).

We always exit with an answer as to whether there was an equal sign following a name.

3.34.1.4 DATA Expression Scan (DEXPSCAN)

This is used for scanning the expressions used as subscripts or DO control expressions within DATA statements. It has the same hierarchy structure as regular expression scan, starting with plus and minus, then multiply and divide, and then expression elements. No exponentiation is allowed, and the only permissible elements are DO control variables, integer constants, and nested expressions. When a variable name appears, there are several things we have to do with it. We do not care what it has been used for by the outside world, as long as it is integer. We call Check for Inner DO Control, to make sure that the name has not appeared as an inner DO control variable. Then we see whether it has already been registered on the DSUBLIST. If so we return with a pointer to it on that list. Otherwise, we create a new entry for it, as described above.

There is a small difference between the plex notation used for DATA subscript expressions and other expressions. In ordinary expressions, a sum or product may have two or more terms; in DATA subscript expressions, they have exactly two terms. Thus J+K+L is represented as +(J,K,L) in an ordinary exprssion (permitting optimization), whereas, in DATA, it is represented as +(+(J,K),L). This simplifies the DATA expression scanner and the expression evaluator.

3.34.1.5 Check DATA Subscript Variables (CHKDSUBS)

After one variable list and constant list have been scanned, this routine is used to step through the DATA Subscript List looking at each of the variables that has appeared in either a subscript expression or DO control expression. Each of these should also have appeared as a DO control variable. (Only DO control variables are allowed to appear in such expressions.) If it has appeared as a DO control variable, the third word of the DSUBLIST entry will reflect that fact. If it has not, we put out a diagnostic, marking it under the variable name. A character count has been saved for us for this purpose in the second word of the DSUBLIST entry.

3.34.1.6 DATA Constant List Scan (DCONLSCN)

Used to scan the constant lists (between slashes) in a DATA statement (or type statement). It pules up numerous constant items on the Term List, and then builds a DATA Constant List Plex from them. Some of the items may have multipliers ahead of them. The way we handle this is to scan the multiplier as if it were a constant item. The, when we discover an asterisk after it (or a left parenthesis implying a multiplier), we pull apart the Character Count Plex that was built around it and make sure the constant item was an Integer constant (no other type of multiplier is allowed) and that it is a reasonable size. Then we can scan the constant item that the multiplier is to be associated with.

A DATA Constant Item is generally any kind of constant, and is scanned by Get Constant, using the entry point GETCONB, which allows long hexadecimal and alphanumeric strings. The resultant constant pointer is combined with the character count at the last character of the constant, and a Character Count Plex is made from them. If a left parenthesis is found, we y;simply;y recurse back to DATA Constant List Scan. This has the effect that a Complex constant will be treated as a sub-list of two items -- nothing complex about it -- and this is perfectly all right; the expander will work with separate constants just fine.

The situation becomes more complicated when a preceding plus or minus sign is encountered. This forces the constant to be numeric; it cannot be Logical or Hollerith or Hexadecimal. However, it can be Complex, unfortunately. So, if we find a left parenthesis following a plus or minus sign, we TRY for Complex constant. There is nothing else it can be, so we force the TRY to either succeed or to fail completely (i.e., to not return with answer FALSE). If a left parenthesis did not follow the plus or minus sign, then we can just call Constant Scan and allow Integer, Real, or Double Precision constants only. Then if it was a minus sign, we call Negate Constant to negate it.

3.34.2 Expand DATA Pair Sequence EXPDPRSQ)

A DATA Pair consists of a variable part and a constant part. The definition of each part is recursive, and either part may be a nested structure. The variable part may be a simple variable, or a list of variables with or without DO control, or a list of the above, etc.; the constant part (which appears between slashes in the source statement) may be one constant or many, or a repeated constant, e.g., 5*3.14159, or a repeated list of constants, e.g., 3*(1.78,-6.2E3,.TRUE.,23), or any of the above nested to any depth.

Expand DATA Pair Sequence receives these varaible and constant things as structures represented by plexes. Its job is to expand them from the plex form into lists of variables and constants and to interface these lists. It would be possible to expand the variable list into memory and expand the constant list into memory, then interface the two lists an item at a time. The only trouble would be that such an expanded list may be thousands of words in length, and we would hate to run the compiler out of memory. Therefore the expansion is done alternately, one variable item and one constant item at a time; the method strongly resembles the run-time processing of an input statemnt via a FORMAT statement; control passes back and forth between the variable processor, which needs constants to satisfy its variables, and the constant processor, which needs variables to satisfy its constants. Each of these two processors maintains the illusion that it is in control and calls the other as a subroutine. The two key routines in the double recursion are Get Constant (GETCONST) and Use Constant (USECONST); each of these simply switches exit lists, then exits. (Some clever initialization exits so that these routines will work the first time.) GETCONST is called by the variable processor, and USECONST is called by the constant processor.

The constant processor is quite simple and only 47 POPs long. This is because the structure of constant lists is simpler than that of variable lists, and because all the error checking is done by the variable processor. The latter is 225 POPs long and is complicated. Quite a lot of error checking goes on. Implied DO loops can run amok. Variables and constants may be of incompatible type (mode). Subscripts may get out of range, and so forth. The convention about errors is that if anything goes wrong with a Data Pair, the whole Data Pair is deleted. Because every time we successsfully match a variable with a constant we put it on the OUTLIST (and hence potentially to the scratch file between Pass 1 and Pass 2), some trickery is needed to allow us to deleted Data Pair retroactively. The Data Pair Size List (DPRSZLST) is kept in memory and allows Pass 2 to delete a Data Pair that Pass 1 has found faulty. The gimmick is this: every time Pass 1 puts out some words for a Data Pair, it keeps track of how many it put out; this is done in a temp called DPRWDCNT (DATA Pair Word Count). If something goes wrong with the data pair, we append this count to the DPRSZLST. Thus when Pass 2 receives a Data Pair Number (DPAIRNO), it removes the top item from the DPRSZLST and then ignores that many words in the input stream. If Pass 1 gets through a data pair without errors, it puts a zero on the bottom of the DPRSZLST, thus telling Pass 2 to ignore no words. See DATA Pair Gen, 3.34.3.3.

Let us examine Expand DATA Pair Sequence (EXPDPRSQ) from the outside in. The outer routine is thirteen words long. It receives a sequence of data pairs, and this sequence is represented as a file on the WORKLIST. This file is simply variable item, constant item, variable item, constant item, etc. (Each item may, of course, be a nested structure.) EXPDPRSQ FADs a Begin Data Number (BEGINDNO) then moves its file over to the DPAIRLST, retaining a reserve mark on the WORKLIST. It then initializes the Total Iteration Count (TOTITCNT) to 131072, the largest possible core size. This count gets reduced every time a variable list item is processed. This is a rather large upper bound, but it can keep the compiler from looping endlessly on a DATA statement that tries to do too much. Next we go into a short loop of removing two-word items from the DPAIRLST and JRSing to Expand Data Pair (EXPDPAIR). After the loop, we FAD an End DATA Number (ENDATANO), move the code from the CODELIST to the OUTLIST, and exit.

EXPDPAIR (which is called repeatedly by EXPDPRSQ) is concerned with a few error messages and with the housekeeping details for accomplishing the double recursion. It is 30 words long, but it calls routines adding up to 284 words. EXPDPAIR begins by outputting a Data Pair Number. It zeroes the Data Pair Word Count and MEFs the CLUTRIG and CLOUTRIG (Constant List Used Trigger and Constant List Overused Trigger). These triggers govern error messages about whether the constant list has been used at least once and whether it has been used more than once. There is currently a bug concerning the CLOUTRIG, and the error message "WARNING: TOO FEW CONSTANTS; LIST REPEATED THRU HERE" never comes out. Next we create a file of exits on the PARAMLST. These exits are for expanding the constant list, and this file will be on the EXITLIST whenever constants are being processed. The list is intialized so that the first time GETCONST is called it will exit to OVERNOVR (Process Constant List Over And Over). Having finished with the Constant exit file, we move the constant pointer from the work bottom to the bottom of the DACONLST. Then we reserve the EXITLIST and prepare it to handle the recursion for the variable list. The only reason we have to reserve it is that in case we discover an error several recursion levels down, we can get out of trouble by releasing the EXITLIST.

Because of a peculiarity fo the Answer Flag mechanism, we create a fake EXITLIST entry of 1. This allows answers to be returned at level zero of this routine. Now we are ready to JRS to PDVITEM (Process Data Variable Item) which is where the real work is done. After returning, we JRS to GETCONST (Get Constant). We do not really need or want this constant. It should be the constant after the last constant in the source statement--i.e., it should be the the first constant provided by the automatic wrap-around feature of constant lists. The reason we call GETCONST is to set the CLUTRIG if we have indeed used up all the constants. We then TMT the CLUTRIG and conditionally put out the message "WARNING: CONSTANT LIST NOT COMPLETELY USED." The error mark will appear on the first constant that was not used. Now we are almost done. We replace the unwanted constant pointer with a zero, then fall into Data Pair Exit (DPAIREX) which MONs the DPRSZLST, releases the EXITLIST and PARAMLST, ZAPs the DACONLST (DATA Constant Lit), DREPLIST (DATA Constant Repeat Count List) and DVARLIST (DATA Variable List), and exits.

3.34.2.1 Process DATA Variable Item PDVITEM)

A Data Variable Item will be one of the following five things:

1. DVARLSPX - Data Variable List Plex (this is simply a set of data variable items glued together in a plex; each item is one of the five kinds.)

2. DVLWDPX - Data DO Control Plex

3. ARRAY - Unsubscripted array name (meaning whole array)

4. SCALAR - A simple scalar, lowly born

5. CHCTPLEX - A Character Count Plex attaching a character count to an array element. The array element will be represented by one of the following:

    a. CONSUBPX - Constant Subscripted Array Plex, or

    b. FULSUBPX - Fully Subscripted Array Plex.

PDVITEM starts off by MMDing the Total Iteration Count (TOTITCNT) and checking to see if it has gone negative. If it has, we put out the message "WARNING: MORE THAN 131,072 CONSTANTS GENERATED. DATA GROUP DELETED." Then we go to DATAFAIL, which sets the DPRSZLST so that Pass 2 will delete the data group.

Assuming that the above error did not occur, we try to figure out which of the five kinds of variable items we have here. If it is a Data Variable List Plex, it is easy to handle. We simply RPL to the DVARLIST and take the items off the top one at a time and JRS right back into PDVITEM with them.

If the item is a Data DO Control Plex, the processing is a little stickier but still takes only 25 POPs. We pull the plex to the WORKLIST, thus putting 6 words there, as follows:

W5 = Data Variable Item controlled by this DO (one of the five kinds listed above)

W4 = DO Control Variable (Pointer to DSUBLIST)

W3 = Character count for "=" in DO (to guide the placement of an error mark)

W2 = Initial Expression Pointer

W1 = Terminal Expression Pointer

W0 = Step Expression Pointer

The three expressions need to be evaluated, and this is done for each by JRSing to EVDATSCR (Evaluate DATA Subscript Expression) which is called with an expression pointer in W0 and returns with the numerical value in W0. Some twiddling of the bottom three items of the WORKLIST is done as these expressions are being evaluated, and the initial value and step value are stored into the DSUBLIST. Then we QBL to a machine language routine that calculates the number of iterations that this DO will require. The routine is called CALLDOCML (Caluculate DO Count ML) and expects the initial vaue in W0, termination value in W1, and step size in the A register. It discards W0 and W1 and returns the iteration count in INTCENT. It gives the answer .TRUE. if there is to be at least one iteration, .FALSE. otherwise. If the answer is .FALSE., we simply discard three items from the WORKLIST and exit; our DO is done. Otherwise we pick up the iteration count, check to see if it is too big (over 131072) and store this count into the DSUBLIST. If the iteration count is too big, we put out the message "WARNING: DO-COUNT TOO LARGE. DATA GROUP DELETED." This message flags the equal sign in the DO-control part of the source statement.

Assuming that we got by without error, we are now ready to do the DO loop. The item pointer controlled by the DO is now in W1. (W0 contains a pointer to the DSUBLIST for the item we are supposed to step). After returning from that one iteration we QBL to Step Data DO Loop ML (STDADOML). STDADOML returns an answer .TRUE. if another iteration is called for. STDADOML also increments the DO control variable so that subscripts using it will be up to date. And that is all there is to DATA DO loops.

If the Data Variable Item is not a Data Variable List Plex and is not a Data DO Control Plex, perhaps it is an unsubscripted array. We must break the array down into its individual elements; the Pass 1/2 interface data will thus look just as if the user had written out all the elements of the array in storage order. We FET a zero which is the initial addend, then JRS to a routine called Compute Number of Elements and Size From W1 (CELSIZW1) which comes back with four things on the WORKLIST:

W3	=	Array Name	(Pointer)
W2	=	Current Addend	(Value, initially zero)
W1	=	Number of Elements	(Value)
W0	=	Element Size in Words	(Value)

For each iteration we must add the element size to the current addend, and SKR the Number of Elements. To handle each element, we JRS to a routine called Use Addend And Array Name (USEADARY) that is also used for array elements with explicit subscripts. This is called with the addend value in W0 and the array name pointer in W1. When the array has been exhausted we clean up the WORKLIST (discard 4 items) and exit.

Now we are getting close to the nitty gritty. If the variable item is not any of the three things discussed on the last few pages, it is either an array

element (with character count attached) or a scalar. The array element case is only a little more work than the scalar: first we PUL the plex to the WORKLIST and STO into AYELCHCT (Array Element Character Count), this count being retained in case the subscript is out of range. Then we PUL again to the WORKLIST, this time pulling either a CONSUBPX (Constant Subscripted Array Plex) or a FULSUBPX (Fully Subscripted Array Plex). We have SOTed before the PUL to determine which. Now we have either one or two subscript expressions to evaluate. The subscript in a CONSUBPX and the constant part of a FULSUBPX are represented by pointers to the Integer Constant List, so evaluating them simply amounts to fetching the values pointed to. In either constant or variable case, a JRS to EVDATSCR (Evaluate Data Subscript Expression) is involved, and two calls are involved for the FULSUBPX. We wind up with a subscript value (in words, not elements) in W0 and an array pointer in W1 and we have only to check to see if the value is in range (OJS to ISSVIR), and if it is we fall into Use Addend And Array Name (USEADARY, described above). If the subscript is out of range, we put out the message: "WARNING. SUBSCRIPT NOT WITHIN ARRAY. DATA GROUP DELETED" and we go to DATAFAIL.

USESCALR (Use Scalar) and USEADARY (Use Addend and Array Name) are two entry points to essentially the same routine. They combine after three POPs. USESCALR starts out

```
        FAD     DITEMNO     DATA Item Number
                            FIL     =0          File Addend for
Scalar
                    5G1     FIL     W0          File ORG
```

and is joined by USEADARY at 5G1. USEADARY has begun

```
                            FAD     DITEMNO     DATA Item Number
                            MON     CODELIST    File Addend for Array
                            BRU     5G1
```

From now on it does not matter whether we have a scalar or an array element. What we have is called a Variable, and it is important whether said Variable is of type complex or komplex or neither. Non-complex variables are satisfied with single constants; complex and komplex variables sometimes take more than one constant. The using of the Variable we now have will be done JRSing to one of three routines: USECMPXV, USEKMPXV, or USENCPXV. These routines all call GETCONST and are quite concerned as to whether the constant gotten is satisfactory for the Variable.

Of the three routines just mentioned, USENCPXV (Use Non-Complex Variable) is 53 POPs long, USECMPXV (Use Complex Variable) is 23 POPs, and USEKMPXV (Use Komplex Variable) is 20 POPs. The complex and komplex routines are shorter not because the rules for complex and komplex are simpler, but because they get some of their work done by calling USENCPXV. On the 9300, the rules for

what constant could go with what variable were quite simple. On the Sigma, with two new additional constant types (hexadecimal and string), the rules are just plain horrible.

Use Non-Complex Variable, if the constant is not hexadecimal and not string, the thing falls out in four POPs (out of a total of 56). This fact is a little misleading, however, since one of the POPs is QBL to ISMCNVML (Is Mode Convertible ML), a little machine-language routine that does a table loopup and takes 15 words. The table allows conversion from IRO to IRDCK, from CK to CK, and from L to L, where the letters stand for integer, real, double, complex, komplex, and logical.

In Use Non-Complex Variable, suppose GETCONST gives us a string constant. Then what it has in fact given us is one word of a string constant. We do not care if the word is the whole constant or part of a longer constant, and GETCONST gives us no means for distinguishing. We do a GRP 0 which gets the value of the constant; this we store in CENTRAL1 and in INTCENT. Now we check whether the Variable (whose pointer is in W0) is double. If it is, we JRS to GETCONST again and require that the new constant also be string. If it is, we GRP its value and store it in CENTRAL2 and JRS to RGDBLCON (Register Double Constant) and BRU to 6G1, a rather common exit point that goes

```
6G1     MON     CODELIST

        BRL     EXITML
```

If the following constant is not string, we go to DMODFAIL (Data Mode Fail) which marks the constant and gives the message: "WARNING: CONFLICTS WITH VARIABLE TYPE. DATA GROUP DELETED." (Incidentally, the phrase "DATA GROUP" in an error message means the same thing as "Data Pair" in the internal terminology.)

We are tstill talking about non-complex variables and string constants, and we have just finished the case of a double precision variable. If it is not double, it must integer, real, or logical; if it is integer, we JRS to Register Hollerith Constant (RGHOLCON). This is for a single-word constant of type integer. If the variable is real, we zero CENTRAL2 and JRS to Register Real Constant (RERLCON). If the variable is logical, we JRS to Register Logical Constant (RGLOGCON) which actually registers the thing on the Integer Constant List (ICONSLST) and changes the pointer's mode to logical. Now, no matter what kind of constant we have registered this string constant as, we have performed no arithmetic conversion on it, and we BRU to our familiar exit 6G1 described in the previous paragraph.

We are still talking about non-complex variables, and the only constant type remaining is hexadecimal. Hex constant in DATA statements are rather peculiar since, unlike other places, they can be up to 32 characters in length. Also unlike other places, they are registered on the DHEXLIST (DATA Hexadecimal Constant List) rather than the XCONSLST (Hexadecimal Constant List). On the XCONSLST things are of type integer; on the DHEXLIST they are of type unknown. Each is a five-word item. The first four words are the value of the

constant--right-justified with leading zeroes. The fifth is a word count, indicating how many of the four words are significant. In the non-complex case, a word count above 2 is not allowed. If a longer constant is written, we go to DHEXFAIL which produces the message "WARNING: HEX CONSTANT WRONG LENGTH. DATA GROUP DELETED." The last character of the constant is marked.

If the length is 2 or less, we get the 4th word of the constant and store it into INTECENT, CENTRAL1 and CENTRAL2--all the places it might be needed for registering. Then we check the mode of the variable. If it is double precision, we demand the constant be 2 words in length. We get the 3rd word of the constant and store it into CENTRAL1. Then we register the constant as double, then BRU to 6G1 where the pointer goes onto the CODELIST, and we exit. If the variable is real or integer, we demand a length of 1 or less and BRU to 6G3, which registers single-word constants in the string case. Incidentally, in the integer case the hex constant gets registered as Hollerith. This is wrong, but it causes no trouble because we never print the output of a DATA statement. The value is output correctly in binary. If an option were ever installed to have LO list the expansion of DATA statements (an easy thing to install) we would have a problem here.

Now we come to the complex and komplex variable cases. The trouble with these is that they may take one or two constants, depending on the kind of constant. We are now discussing routines called Use Complex Variable and Use Komplex Variable (USECMPXV and USEKMPXV). Each calls GETCONST and compares the constant mode with the variable mode. If they agree we rejoice and go to 6G1. If they do not we curse and check the constant for hex. If it is not hex, there is no chance that this variable can be satisifed with a single constant. So we fake up a variable pointer of either real or double mode, put it in W1, and JRS to USENCPXC, then to USENCPXV. The former expects that the constant has already been gotten; the latter begins with a JRS to GETCONST. Thus we satisfy the complex or komplex variable as if it were two real or double variables.

In USECMPXV, if the constant is hexadecimal, we check the length. If it is over 2, we go to DHEXFAIL. If it is exactly 2, we move the 3rd value word into CENTRAL1 and the 4th into CENTRAL3. We zero CENTRAL2 and CENTRAL4, register the constant as complex, and BRU to 6G1. If the length is less than 2, we fake a real variable and JRS to USENCPXC and USENCPXV, behaving as if there were two real variables.

In USEKMPXV, if the constant is hexadecimal, we check the length. If it is over 2, then this constant satisfies the whole variable. We put the 4 value words into CENTRAL1 through CENTRAL4, register the constant as komplex, and get out through 6G1. If the length is 2 or less, then we go through the two-constant route as if there were two double variables. Incidentally, since a double can be satisfied with two string constants, a komplex variable can be satisfied with four. (Each string constant may be a one-word piece of a larger string constant.

3.34.2.2 Process DATA Constant Item (PDCONITM)

Now we come to the other side of the double recursion. The PDCONITM subroutine is actually driven by Process Constant List Over And Over (OVERNOVR). The interface with the outer world is through Get Constant (GETCONST, called from PDVITEM) and Use Constant (USECONST, called from PDCONITM). These routines simply swap Exit Lists, then exit, thus maintaining the fiction that both sides are in control.

A Data Constant Item will be one of the following three things:

1. REPCONPX - Repeated Constant Plex which when PULled yields a repeat count constant pointer (to ICONSLST) in W1 and the item to be repeated in W0. The item to be repeated may be a constant (with a character count plex attached) or a Constant List Plex or a Repated Constant Plex.

2. DCONLSPX - Data Constant List Plex which is simply a set of data constant items glued together in a plex. Each item is one of the three kinds.

3. CHCTPLEX - Character Count Plex attached to a constant pointer. The character count is useful for flagging errors. W1 is the constandt pointer, an W0 the character count.

3.34.3 DATA Statement Passes 2 and 3)

95 per cent of the work of the DATA statement is done in Pass 1, so Passes and 2 and 3 have a pretty easy time of it. Nonetheless, there are four statement types in the Pass 1/Pass 2 interface for the DATA statement, and three in the Pass 2/Pass 3 interface, so there must be something going on. The main things that Pass 2 does are to make the constant mode agree with the mode of the variable (Pass 1 makes sure that the modes are compatible, but does not actually convert any constants) and to ignore the output of any defective DATA statement.

3.34.3.1 Begin DATA Gen BDATAGEN _ Pass 2 _

Pass 1 generates this indicator at the beginning of a DATA statement. Pass 2 could not care less and goes directly to STEXNSA (Statement Exit No Scrip Gen or Assembly).

3.34.3.2 End DATA Gen EDATAGEN _ Pass 2 _

Pass 2 generates a Return-To-Program-Location-Counter Indicator which is needed because each constant in the DATA statement has had its own ORG.

3.34.3.3 DATA Pair Gen DPAIRGEN _ Pass 2 _

This title is misleading, for this routine does not gen anything but rather assures that certain things will not be generated. A DATA pair is a variable and its associated constant list (enclosed in slashes). The expansion of such a pair in Pass 1 may produce many thousands of words of interface data. If at any time during the expansion of a DATA pair an error occurs (such as a subscript going out of range or a constant having a mode incompatible with the variable), we want to ignore the whole DATA pair, and we want ot do this without Pass 1's having to hold all the data for the whole DATA pair in core at once. So Pass 1 sends the interface data off one item at a time, but keeps track of how many words it generates. If the data pair goes bad, it appends this count to the DATA Pair Size List (DPRSZLST); if the pair is valid, it appends zero. This is a count of how many words Pass 2 should ignore, and there is one such count for every data pair in the source program.

So when Pass 2 gets a Data Pair Indicator, it takes the top word off the DPRSZLST, then JRSs to NXINWORD that many times (possibly zero) and throws away that many words of interface data.

3.34.3.4 DATA Item Gen DTITEMGN _ Pass 2 _

This statement driver is the only one whose Pass 1/Pass 2 interface is of variable length and whose length is not indicated by a count. It is either four or five words in length, and the longer version occurs only when a complex or komplex variable has been satisfied with two non-complex constants rather than a single complex or komplex constant. This is a more common occurance than it might seem, for in the statement DATA C /(2.5,4.7)/ Pass 1 sees two real constants rather than one Complex one. Only if the constant has a leading plus or minus sign, as in DATA C /+(2.5,4.7)/ will Pass 1 see it as a complex constant. So the first DATA statement above produces five words of interface data, while the second produces four.

The four (or five) interface words are as follows:

1. Indicator
2. ORG Addend value
3. ORG Symbol pointer
4. Constant pointer
5. Imaginary Part constant pointer

The second word gets masked to machine address size and its upper byte changed to a Data ORG With Addend indicator. This is put on the Code List. Next the third word goes onto the CODELIST just as it is, then the fourth word (the constant pointer) is gotten and its mode is compared with that on the bottom

of the CODELIST. If the mode agrees, we file a Data Constant indicator, then file the constant pointer.

If the mode disagrees, there are two possible reasons why: there may be a complex/non-complex problem, in which case we have to amalgamate two constants into one; or it may be an "ordinary" one-to-one mode disagreement, in which case the constant must be converted. Amalgamating is done by Convert Pair To Complex Constant (CVPAIRCC) which is in PASSOF. Other constant mode correction is done in Correct Constant Mode (CORCONM) in Pass 2. In any case, the output to Pass 3 is one constant pointer of exactly the same mode as the variable.

3.34.3.5 DATA Statement in Pass 3

There are three Pass 2/Pass 3 interface items for the DATA statement, and three routines in OUTPORG to handle them:

1. OUTDTORG (Output DATA Org) is given a pointer and offset to the variable into which the data is to be loaded. It calls EVALPNTR and BOORG (BO Origin) to generate the appropriate object language bytes.

2. OUTDCON (Output DATA Constant) is given a constant of any mode. If that mode is real or complex, OUTDCON calls CDTORML (Convert Double to Real) or CKTOC (Convert Double Complex to Complex) to change the given Long Floating Point value(s) to the properly rounded and truncated Short Floating forms. OUTDCON calls BODATUM (see 2.28.2) once for each word of the constant.

3. OUTRPROG (Output Return to Program Location) simply calls BOORG (see 2.28.2) with the current value of the Program Location Counter (LOCCOUNT) in order to generate the object code origin back to the next program location.

These routines do not call any of the object listing generation routines since DATA statements produce no object listing.

3.35 DECODE Statement

3.35.1 DECODE Statement (DECODEST - Pass 1)

This statement is exactly like ENCODE, except that the first word filed is DECODENO instead of ENCODENO. The work is done by Encode-Decode (ENCODECO), under ENCODE Statement (Section 3.41.2.1).

3.35.2 DECODE Statement Gen (DECODESG - Pass 2)

DECODE Statement generation is much like ENCODE, and the two routines share all but three POPs of their code. See the writeup "ENCODE and DECODE Statement Gen", Section 3.41.3.

3.36 DIMENSION Statement (DIMENSTA - Pass 1 only)

Nearly all of the work is done by the subroutine Allow Dimensions, so that this can be used by all the other statements in which dimensions may be declared (e.g., COMMON, REAL, INTEGER). The only difference is that in those statements, a variable need not be dimensioned; in a DIMENSION statement, it must be. So we go in through a special entry called Register Dimensions, instead of the usual entry that exits if a left parenthesis is not present. Exit is through Declarative Exit 2 (valid in BLOCK DATA).

No information is transmitted directly to Pass 2. The information is collected on the Symbol Table, Array List, and Group List.

3.37 Allow Dimensions (ALLOWDIM)

Used by the DIMENSION statement, and also by other declarative statements in which variables may be dimensioned (e.g., COMPLEX, GLOBAL). In the latter case, if there is no left parenthesis following the name, the routine exits immediately.

First we check for a variety of error conditions. If the name is not permissible to be an array (see Permissibility, 2.8.1), or has already been declared as an array, we put out a diagnostic and skip over the dimensions listed without processing them at all. This allows us to go on with the rest of the statement. If there is no left parenthesis following a name in a DIMENSION statement, but there is a comma or end-of-statement, then we can assume that the user forgot to include the dimensions, and we tell him so and ignore the name. If there is something other than a comma or end-of-statement, then there is something drastically wrong, so we just go off to Illegal Syntax.

Assuming there are no illegal conditions, we set up new permissibility for ARRAY, but we do not register that class yet. Temporarily (until we have processed the dimensions) we register it as ILLEGAL. This is not really necessary, but it falls out of a couple of other things: to prevent statement functions from calling themselves, the function name is registered as ILLEGAL before processing the expression. Thus, for example, the statement:

```
F(X) = X + F(Y)
```

will be diagnosed on the second "F" as being a reflexive (i.e., recursive) definition. This mechanism is all there in the part of expression scan that

scans identifiers. Secondly, once upon a time we were thinking of allowing expressions in DIMENSION statements. We have to do expression evaluations anyway with adjustable dimensions. There is no reason we could not allow constructs like:

```
SUBROUTINE SHAZAM(K,N,M,L)

DIMENSION K(N-2:M*L+N-3)
```

One of the reasons we never did was that it did not add much, except complication. Also, the 9300 FORTRAN method of indicating upper and lower bounds was with a slash, which would be indistinguishable from division, and we wanted to be able to accept old 9300 programs. However, had we allowed expressions, consider the curious situation produced by the statements:

```
SUBROUTINE SHAZAM(K,N)

DIMENSION K( K(2,3) , N )
```

To know what the first dimension is, you have to know what is in K(2,3). But to find K(2,3), you have to know what the first dimension is. This is very much like the recursive call shown above for statement functions, except that the word "recursive" is not quite appropriate. By using the word "reflexive", we can use the same error message for both cases.

Also save the pointer to the array in DIMTEMP, so that, if we find adjustable dimensions, we can pick it up and make sure it was a dummy. And initialize the Overflow Trigger to FALSE, so that we can test for overflow in the arithmetic operations that follow.

Before proceeding, it would be wise to read the section on arrays, so that you understand what it is that this routine is trying to create on the Arrray List and Group List.

For the following discussion, assume that the dimensions specified are L and U, as follows:

DIMENSION A(L_1 : U_1 , L_2 : U_2 , ...)

In each case, if no lower bound is specified, it is assumed equal to one. L and U are not necessarily identifiers; more likely they are Integer constants, but we have to prepare for the general case where they are scalars.

As we scan multiple dimensions (single dimensions are just a simple case that falls out of this), we are going to accumulate a number of things. On the DIMLIST, we accumulate multipliers. These specify the size of each row, column, etc. and are equal to (U-L+1) or just (U). If any multiplier is not directly addressable in integer (i.e. if it is not an integer constant or a single integer scalar), there will be an expression plex representing it. For each such case, a two-word entry is created on the TERMLIST. The first word is the expression plex pointer, and the second is a pointer to a private

integer temp. Then, instead of putting the plex pointer on the DIMLIST, We put the private temp pointer there. These multipliers are used whenever an array appears with a subscript. Usually the multipliers are constant and all the work is done at compile time. However, the multipliers can be variable (adjustable), in which case the object code will have to go through the mechanics of multiplying these multipliers. So, each of the complicated expressions will be computed at the head of the program and its value stored in the temp we just created. Then this temp can be used at each reference thereafter. Therefore, we want to keep the directly addressable things around on the DIMLIST, and this means the temp instead of a complicated expression. Only the first N-1 multipliers are needed in subscript calculations. However, since some types of data require more than one word of storage, we create a first multiplier which gives the element size. This starts out as the value one, and stays that way for all the time that Allow Dimensions is working on it. Later, Set Element Size will update it if necessary.

So we have N multipliers collected on the DIMLIST. These multipliers are also used in computing the total size of the array. The size is carried along in WO, and each time a new multiplier is created, we multiply the old size by the new multiplier. Some of these multipliers may be complicated, of course. If they are, we use the expression plex pointer (rather than the temp pointer) in computing the total size. Thus we end up with (possibly) one big combined expression plex representing the size of the array (in elements, so the element size multiplier is not used).

On the SCREXLST (Script Expression List) meanwhile, we accumulate N lower bounds. The lower bound is L, which is 1 if not specified. These will be used to compute the array's offset, which tells how far the actual beginning of the array is from the zeroth element (i.e., A(0,0,0,...)). By collecting lower bounds on the SCREXLST, we are able to make use of the already existing subroutine, Derive Offset, that is used whenever an array is referenced. When it is referenced, it appears with a set of subscripts, and we want to know how far that particular set of subscripts is from the zeroth element. If every subscript is set to its lower bound, that <u>is</u> the actual first word of the array. So we set every subscript to its lower bound, and then use Derive Offset, to tell us how far that is from the zeroth element.

In computing the values such as (U-L+1) and the total size, we make use of several subroutines in PASSOF: Negate Term (NEGTERM), Add 2 and Simplify (ADD2SIMP), and Multiply 2 and Simplify (MUL2SIMP). These routines do the indicated operations on the bottom or bottom two pointers on the Work List. If these are constant pointers, it computes the new constant and returns a pointer to that. If they are expression pointers, it combines the expressions and simplifies them wherever possible (i.e. by combining constant operations), and returns a new pointer to the plex containing the combined expression. In most programs, these numbers are all constants, and the work of these routines is considerably simpler.

If any dimension found is a real or double precision scalar, rather than integer, a Fix Plex is attached to it. This means, of course, that it is no longer simple or directly addressable.

After all the dimensions are processed, we have the size on the Work List, the multipliers on the DIMLIST, the lower bounds on the SCREXLST, and on the TERMLIST we have two-word entries representing complicated multipliers that have to be evaluated sometime and stored in their specified temps. Pass 2 will take care of that. First we save a complete copy of our file of multipliers over on the TERMLIST, because Derive Offset is going to destroy them when it computes our offset. Into the fourth word of the array's Symbol Table entry, we put a pointer to the Array List entry we are about to create. Then the first two items put onto the Array List are the offset and the size. If the offset is not an integer constant, we create a private temp for it, and keep it around to become the 6th item on the Array List. Now take the file of multipliers (which used to be on the DIMLIST, but we saved it on the Term List) and put it on the Group List. Put the pointer to this new group as the third item on the Array List.

If the array is not a dummy, we are essentially through and can go down and finish up (see below). If it is a dummy, there are two or three more items that have to go on the Array List. First, take the file of two-word entries (representing complicated multipliers) that is now on the Term List, and put it on the Group List, and make a pointer to this new group the 4th item on the Array List.

Next, all the plexes that were created in the process of creating the expressions representing multipliers, offset, and size, need to be copied over to the Group List, and a pointer to this new group becomes the 5th item on the array list. There is a complication here. Originally, the plex list was dumped at the end of every statement. To optimize subscript operations, it is necessary to keep the plex list around across statements. This means we can no longer assume that the Plex List was empty when we started this use of Allow Dimensions. We not only do not want to copy other people's plexes onto our array group, but the other people will not be very happy if we throw away their plexes. We thought we could solve this completely by not allowing dummies to be dimensioned inside a DO loop. To achieve this end, we call Fail If Inside DO Loop, which sees if there is an open DO on the DO List. But it turns out a bugaboo still lurks. When IBM decided to allow DATA in type statements (e.g., REAL X/3.14/), we had to call our DATA processor, which builds plexes that are needed by Expand DATA Pair Sequence at the end of the statement. If, later in the same statement, a dummy is dimensioned, these plexes get lost; e.g., REAL X/3.14/, K(N). So what we do is remember whether the Plex List was empty when we started. If it was, great, we copy it over to the Group List with a CAE. If it was not empty, we copy the Plex List over with a CPY (which, unfortunately, takes with it all that other items we do not need, but it does not hurt anything, except space), leaving it all on the Plex List too. Then we manually reduce the size of the Plex List to what it was when we came in, thus throwing away the new information.

Then, if the offset was not an integer constant (above), we put a pointer to its private temp as the 6th item on the Array List.

Now we can finish up and get out. The array name no longer has to be ILLEGAL, so register it as ARRAY. If the mode is already known, call Set Element Size,

which uses the third item on the Array List to locate the group of multiiliers on the Group List. Then it changes the first multiplier from 1 to whatever the appropriate data size is (which may be 1). If the mode is not known yet, then we bide our time. The element size will get set whenever the mode is declared or the array is used (which forces it to acquire implicit type). (See Register Type, Set Type, and Impose Type, all in Section 2.8.3.)

3.38 DO Statement

3.38.1 DO Statement DOSTA _ Pass 1)

Use DO Statement Number Scan to get a statement number and make sure it is valid in this DO statement. Put this statement number pointer on the DO List, followed by the current actual line number (see Derive Line Count ML), followed by a pointer to the control variable, which must be an Integer, Real, or Double Precision scalar. Allow an optional comma preceding the control variable.

For each DO or REPEAT statement, the DO List receives the above three items. As the range of each DO (or REPEAT) ends, these three items are used to process the end-of-DO (see Process Ends of DOs, PROCENDO, in PASS1F) and check for illegally nested loops. Then they are discarded. Thus there is something on the DO List only while in the range of a DO (or REPEAT). If any loops should happen to never be closed, there will be something on the DO List at the end of the program, which Check For Unclosed DO-loops, under Finish Up, in PASS1F, detects and diagnoses. The line number is passed along so that the DO loop in error can be identified precisely.

Call Outer DO Check to make sure this control variable is not the same as one that is currently in use from an outer DO (or REPEAT). File DOSTANO. Call DO-Control to process the three expressions, file the control variable, and file a DO-control triple plex. Then exit through DO Exit.

Three items are transmitted to Pass 2:

1. DOSTANO
2. Control variable (pointer to symbol table)
3. Pointer to DO-control triple plex, which contains the three expressions

In addition, three items are placed on the DO List, for later use by Pass 1:

1. Terminal statement number (pointer to one of the label lists)
2. Current line number (value, not pointer)
3. Control variable (pointer to symbol table)

Later, there will also be transmitted to Pass 2 either an ENDDONO or an ENDONTNO (for illegally nested loops); see PROCENDO in PASS1F, Section 2.9.7.

3.38.1.1 DO-Control DOCONTRL

Used by DO, REPEAT, and I/O loops. Scans at least two integer, real, or double precision expressions, and puts their pointers on the Work List. If there is no third expression, it creates a pointer to the integer constant 1, and puts that on the Work List. From these three pointers, it builds a Triple Plex and files it.

3.38.1.2 DO Statement Number Scan DOSTANUM

Used by DO and REPEAT statements to get and deal with a terminal statement number for the loop. If the first character encountered is a letter, the user may be trying to use an assigned variable as the DO terminus, which is illegal, but we will put out a diagnostic and try to get through the statement to look for other diagnostics.

Otherwise, call Statement Number Scan (STANUMSC) to scan and register either a local or non-local label. Set up the label trait DOLABEL, which is tested for only in run-time debug mode, by Register Source Line Number (REGSRCLN) in PASS2F. Then, if this statement number has already appeared in the program (i.e., has the label trait DEFINED), give a diagnostic but proceed.

3.38.1.3 Outer DO Check OUTERDO

Called by DO and REPEAT to assure that the current control variable (in WO) is not already in use by an outer loop. This is done with a special Search With Key, since the normal Code word for the DO List is set up to look at the first item in each entry (the statement number), whereas we want to look at the third item in each entry (the ;control; variable pointer). Note that it is not necessary to declassify the identifier pointer (see, for example, NAMELIST Statement), since the first appearance of a variable in a DO (or REPEAT) statement would have determined its class and type, so these cannot have changed. If the variable is found in an outer DO, give a diagnostic and proceed.

3.38.2 DO Statement Gen DOSATG _ Pass 2)

Code generated for the DO statement falls into one of three cases, depending upon the complexity and modes of control variable, step, and termination values. Considering a statement of the form

```
DO 23 V = A,B,C
```

the cases are as follows:

1. If V is not an integer or if C is not a constant, the generated code computes the iteration count (actually, one minus the iteration count) from A, B, and C, placing it in a Private Temp (one which cannot be altered by code inside the loop). The values of V and of the count temp are modified each time through the loop. The generated code behaves as if it were written:

```
        COUNT = INT(((V-A)-B)/C)
1       . . .
        . . .
        V = V+C
        IF ((COUNT=COUNT+1) .LE. 0) GO TO 1
```

The last statement is compiled as an MTW,1 followed by a BLEZ.

2. If V is an integer, and C is constant, but either B is not a constant or C is not +1 or -1, then several savings in object code and run-time are realized by generating code that behaves as follows:

```
        V = N = A
1       . . .
        . . .
        V = N = N+C
        IF (N    b) GO TO 1
```

In the above, N represents a "DO-register", an index register dedicated to the loop, and .LE. (.GE.) in the last statement is used if C is positive (negative). If none of the statements within the loop's range causes the contents of register N to be clobbered, its value will be the same as that of the control variable throughout the loop, thus simplifying indexing when the control variable is used as a subscript.

3. If V is an integer, B and C are constants, and if furthermore C equals plus or minus one (as often happens) then even greater savings are realized by using BIR or BDR to control the loop. In this case, as in the last, a dedicated DO-register is used, but its value differs from that of V by a constant amount, in order to cause the BIR or BDR to fall through at the correct point. The code behaves as follows:

```
        N = (V-A)-B-C
```

```
1    . . .

     . . .

     V = V+C

     IF ((N=N+C)   0) GO TO 1
```

The ( ) is used in the last statement if C=+1 (-1). That statement is implemented as a BIR,N or a BDR,N according as C=+1 or C=-1.

In cases 2 and 3, the compiler remembers what it is using each DO-register for on the DO Register List (DREGLIST). There are three DO-registers; when DO nesting exceeds depth three, they are re-used for the inner loops and the information concerning their use in outer loops is saved on the DOLIST, one four-word file per loop level. Whenever DOSTATG compiles code that makes use of a DO-register, it saves (on the DOLIST) the DREGLIST pointer corresponding thereto, as well as three words from the corresponding DREGLIST item: its Control Variable Rung (YDOCVAR), Addend Rung (YDOADND), and Level Rung (YDOLEVEL). The DREGLIST item is then modified to reflect information concerning its new use: a pointer to V is placed in its Control Variable Rung, zero (case 2) or -B-C (case 3) goes to its Addend Rung, a pointer to the DOLIST (at a point corresponding to the location of the saved information) goes to its Level Rung, and its Clobbered Rung (YDOCLOB) is set FALSE as an indication that the DO-register contents are currently intact. When the End of Loop comes along (see ENDLOOPG, 3.38.3), the latest file of information on the DOLIST is removed therefrom and used to restore the DREGLIST item it points to. When DOSTATG generates case 1 code, it does not tamper with the DREGLIST, and it places an empty file on the DOLIST so that ENDLOOPG will have something to remove; there is a one-to-one correspondence between DO/REPEAT loops and files of save information on the DOLIST at any given time during Pass 2.

3.38.3 End of Loop Gen ENDLOOPG _ Pass 2

The same interface item, with its corresponding gen routine (ENDLOOPG), is used to end all kinds of loops:

- DO-loops
- REPEAT FOR loops
- REPEAT WHILE loops
- implied-DO loops in I/O lists

The corresponding statement gen routines (DOSTATG, REPWHSG, REPEATSG, and IOOOGEN) create two files to be used at end-of-loop time:

1. A file of code on the Delayed Code List; this is the code to step and test the loop; and

2. A file of information on the DO List (DOLIST); this information tells the compiler how to restore a DREGLIST item for outer loop levels.

ENDLOOPG does the following:

1. It restores (by calling RESTORDO) those DO-regs which are in use and currently "clobbered" (see 2.17.5). This is done in order to satisfy the compiler's assumption that, at the top of the loop, the DO-reg contents are "in".

2. ENDLOOPG then examines the bottom file on the DOLIST which is either empty or consists of a four-word DREGLIST save item, as described in 3.38.2 and 2.17.2. In the latter case, ENDLOOPG calls CLOBINDX to wipe out the XREGLIST contents (mainly because the plex pointers in the YINDWHAT rungs thereof will be invalid relative to the new plexes that will be coming in) and then restores the contents of the DREGLIST item pointed to by the first (bottom-most) word on the DO List. The bottom DOLIST file is released.

3. If there was no DREGLIST restore item on the DOLIST (i.e., that file was empty), and there is a nonempty file of code on the Delayed Code List, then we call CLOBALL to forget the contents of all X-regs and DO-regs because we do not know what effect the delayed end-code will have on those registers. If there had been a restore item, then we will have already clobbered the X-regs and set the DO-regs the way we wanted them (see the previous item, 2).

4. Finally, ENDLOOPG copies the Delayed Code to the Code List and exits.

3.39 DOUBLE COMPLEX Statement DBLPRCST _ Pass 1 only)

See Type Statements, 3.80.

3.40 DOUBLE PRECISION Statement DBLPRCST _ Pass 1 only

See Type Statements, 3.80.

3.41 ENCODE Statement

3.41.1 ENCODE Statement ENCODEST Pass 1)

File ENCODENO and go to Encode-Decode (below), used also by DECODE Statement.

3.41.2 Encode-Decode ENCODECO)

Following the statement ID number, three or four things are placed on the Code List. The first is the expression representing the number of characters per "record". This is an expression whose mode is (or can be converted to) Integer. If it is constant, and negative, give a diagnostic. The second is a Format reference: either a statement number, array name, or assigned variable (see Format Reference, FORMREF). The third is either an array name or a variable of any mode (see Array Name Scan, which must be TRYed).

If there is a comma after the third parameter, the fourth must be an Integer variable, whose pointer is filed. If no fourth parameter, use the value zero (not a pointer to the constant zero).

If there is anything following the right parenthesis, go to Input Output List to process it. Otherwise, go directly to End I/O List.

Like the various input/output statements, ENCODE and DECODE in Pass 1 create three separate parts that are sent to Pass 2. The first, which has been explained above is:

1. ENCODENO or DECODENO
2. Number of characters (pointer to expression)
3. Format reference (pointer to statement number, array name or variable)
4. Buffer location (pointer to array name or variable)
5. Characters scanned (absolute zero if none; otherwise pointer to Integer variable)

The second part (which may or may not exist) consists of I/O items, which are handled by I/O Item (3.60.1.3). The third part is an End I/O List (see 3.60.1.1).

3.41.3 ENCODE and DECODE Statement Gen ENCODESG and DECODESG _ Pass 2)

ENCODESG and DECODESG each begin with an ADR of the appropriate library link (9ENCODE or 9DECODE and then branch to a common point. Like other I/O statements, we are only considering the head of the statement; the I/O list items are considered to be separate "statements".

There are five words of information in the Pass 1/2 interface for ENCODE and DECODE:

1. Indicator

2. Record Size in Characters (IRD Expression Pointer)

3. Format Pointer (statement label, array name, array element, scalar, or multiple dummy element)

4. Buffer Start Pointer (array name, array element, scalar, or multiple dummy element)

5. Number of Characters Process Pointer-optional; zero if absent (Integer scalar, array element, or multiple dummy element)

We begin with a JRS to NXINWORD to get Item 2 above, then JRS to GTACUINT (Grntee Integer Unsigned in AC) to get the number of characters expression value in integer into register 9. Next we JRS to FORMGEN to get the Format Pointer and get its effective address into index register 7. FORMGEN can handle all the kinds of pointers mentioned above for Item 3. So far things have been quite straightforward--very much like the BCD READ statement.

Next comes the Buffer Start Pointer, which we wish to send as an effective address in index register 5. The only tricky thing here is to make sure GTEAX5 (Grntee Effective Address in Index 5) does not clobber index 7; we do this by previously calling BUSYX7. Incidentally, GTEAX7 (called in FORMGEN) and GTEAX5 will not clobber register 9. Now we call FREEX7, then build the library link with a BLC *P0, then discard P0.

We are now less than half way through the code for ENCODESG and DECODESG, and we have handled four out of the five words of input data. The sticky item, obviously, is number 5--the optional Number of Characters Processed Pointer. The other three parameters were inputs to 9ENCODE or 9DECODE. This one is an output, and the value is not available until after the call on 9ENDIOL. (It is available after a call on 9IODATA if the last item has its E bit set, for that is tantamount to calling 9ENDIOL.) Therefore, the code for this item is delayed so that it will come out after the 9ENDIOL (see also 3.60.2.3). Before we RSV the CODELIST, we JRS to SCRPTGEN (Subscript Gen) to make sure that any lingering subscripts are output ahead of the statement code. We now check Item 5 to see if it is zero. If so, we are in good shape. We simply call DELAYCOD (Delay Code) which delays an empty file of code, and exit.

Suppose Item 5 is not zero. Then we have to do somehting with this pointer, and the ugly part about it is that it may be subscripted. First we JRS to CLOBALL to forget the contents of all the X and D Registers. Code down here at the end of the statement must not assume register contents that were available at the middle of the statement and are probably now destroyed. The character count value is now sitting in register 9 waiting to be stored. We are concerned whether the thing to be stored into is Fully Subscripted (SOT

FULSUBPX). If it is, there is the remote but definite possibility that the subscript will be not evaluable in an index register (it may be a function call, for example). If so, then generation of the subscript will clobber register 9. So if we have a fully subscripted item (array element or multiple dummy element), we store register 9 away in an integer temp. This is a fairly cowardly thing to do (since the subscript is probably no threat to register 9, and we do not check the subscript to find out), but we do it.* (Note that the store will not take place if the subscript is constant, for then we would have a CONSUBPX rather than a FULSUBPX.) Then we RSV the CODELIST again to mark the spot where we want retroactive subscript code to come out. Now we BRL to REMXDML (Remember X and D Register Status ML) which saves a copy of the status; this status, you will recall, is Clobbered for all registers. This extra copy of the status is for Retroactive Script Gen, which must also know that these registers are clobbered. Normally, this status is saved at the beginning of the statement since normally SCRPTGEN is retroactive to the beginning of the statement. Here, however, it is retroactive only to the latest file mark on the CODELIST. See also Retroactive Subscript Gen, 2.14.4.

All this business about subscripts, by the way, is nearly always useless, for it is very unusual for there to be a bad subscript here. You might wonder why, if any ugly subscripts come along, we do not evaluate them at the beginning of the whole statement, the way we do with ugly subscripts in other statements? The answer is a design decision made in the days of the 9300 (and documented nowhere except in the listing) that in case the ENCODE or DECODE statement should alter variables that turn out to be subscripts, we would rather have the latest value of the subscript.

We are now at the spot where the character count has been stored away in a temp and the subscript code will have been generated. Now we load up the character count again and BAM a STOREINS (Store Instruction) which, because the pointer is known to be integer, will be a STW,9 instruction. We now JRS to SCRPTGEN to generate any ugly subscripts, then unreserve the CODELIST. Thus UNReserve gets rid of the Reserve Mark near the subscript code, not the one that will be used for delaying this whole character count code until after the 9ENDIOL call; the latter Reserve Mark is still intact. Then we JRS once again to CLOBALL to forget the contents of all the X and D Registers since the code just genned is going to be delayed. The code we are about to generate is for the I/O list, and that code will <u>precede</u> the code we have just generated.

Now we are at the label that was jumped to if Item 5 was zero--i.e., if there were no Character Count Pointer. We Delay Code, and BRU to STEXNA (Statment Exit, No Scriptgen) and we are finished.

---

* Note that the store will not take place if the subscript is constant, for then we would have a CONSUBPX rather than a FULSUBPX.

3.42 END Statement

3.42.1 END Statement ENDSTA _ Pass 1)

The END Statement is rather peculiar, in that once the letters "END" are found, no continuation is allowed. To assure this, turn off the Not End Trigger (NOTENDTR) (i.e., this is a possible END statement), and see if the next character on this card is an ENDCHAR (semicolon is not acceptable). This is handled down in Next Active Character (Section 2.2.2), accessed via the CSK POP, which will not look for continuation cards if NOTENDTR is off. If the next character is not an ENDCHAR, go try the END FILE Statement, but turn the Not End Trigger back on so that continuation will work properly.

It is also necessary to have the NOTENDTR back on before going to Fail If Logical IF, because it will fail on a statement like

```
IF (A 0) END
```

and we would not want to leave the NOTENDTR off while it goes off and starts scanning for other things.

Assuming that we are not under a Logical IF, turn the NOTENDTR back on, force the line to be printed even if no LS (see Print if Summary, 2.3.6), call Check Label After IF (CHKLAIF, Section 2.9.9.6), and exit through the special exit ENDEXIT (see Statement Exits, Section 2.9.9).

No information is transmitted to Pass 2 by ENDSTA. Instead, ENDSTA leaves NOTENDTR off so that FINISHUP will send an ENDNO item to Pass 2 (see 2.9.4).

3.42.2 END Statement Gen ENDSTATG _ Pass 2)

The routine called END Statement Gen (ENDSTATG) generates no code but merely sets the End Trigger (ENDTRIG), and branches to STEXNSA (Statement Exit No Scriptgen or Assembly). The main loop of code for Pass 2 generates a statement, then tests the ENTRIG and if FALSE, branches back up to generate another statement.

If we fall through the branch, we do several bits of cleaning up, then exit from Pass 2. The cleaning up includes: JRS to Output Label Constants (OLBLCON) which generates the "B" instructions stored into variables by ASSIGN statements. Then, if the DEBUGTRG is off, we call Name List Gen (NAMELGEN). If the DEBUGTRG is on, we instead call Debug Table Gen (DBTBLGEN), which includes generation of Debug tables, including a complete Name List.) Then we file an End of Program Indicator (ENDINDIC) and call Assemble Code (ASMCODE) for the last time. This gets the Location Counter up to date so that we can store it into PSIZE (Program Size) for printing in the summary. Then we ZAP the PLEXLIST and exit.

3.43 END FILE Statement

3.43.1 END FILE Statement ENDFILST _ Pass 1

The letters "END" have already been found when this statement is reached (see END Statement, Section 3.42.1). If the next letters are not "FILE", go try END LABELS Statement.

Next we may find any number of expressions separated by commas. For each we file ENDFILNO and a pointer to the expression. Note that Integer Expression Scan will allow any expression that can legally be converted to Integer mode; it may not actually be Integer. Exit to Active Exit.

For each expression found, the following pair of words is transmitted to Pass 2:

1. ENDFILNO
2. Unit number (pointer to expression)

3.43.2 ENDFILE Statement Gen ENDFILSG _ Pass 2)

The ENDFILE statement is generated in the same way as REWIND and BACKSPACE statements. Here we ADR the $ENDFILE Link, then branch to common code shared by all three statements. See the writeup for BACKSPACE Statement Gen, 3.26.2.

3.44 END LABELS Statement ENDLABST _ Pass 1 only)

The letters "END" have already been found (see END Statement, Section 3.42.1). The next letters must be "LABELS", otherwise fail.

This is one of the statements (like END) that can do something irrevocable (it prints error messages directly, rather than saving them up for later with POPs like ERL), so we have to be absolutely certain it will not fail before we proceed. This involves two of the checks usually performed in the Statement Exits (see Section 2.9.9), namely Fail If Under Logical IF (which also checks for end of statment) and Fail If BLOCK DATA.

The work is then done by the subroutine described below, which is also used by internal subprograms (see FUNCTION Statement, 3.52.1). Exit through DATAEXIT, since the above two checks have already been made.

Nothing is transmitted directly to Pass 2. The Local Label List is reserved, but Pass 2 could not care less; this affects only Pass 1.

3.45 End Labels Subroutine ENDLBLS)

Used by END LABELS, FUNCTION, and SUBROUTINE Statements.

At this point, we want to close off the current set of local labels and start a new one. So we step through the Local Label List (from the last reserve, if any) and, if there are any undefined labels, print out a warning message listing those labels. Most of this work is done by Search Label List For Undefined Labels (SERLBL). We merely initialize the Label Count to be negative and provide a pointer to the Top of the Local Label List. On return, if there were any undefined labels found, LCOUNT will no longer be negative, and we call Print Error Message 1 (PRERRM1) a couple of times to print the last line and a blank line.

Finally, reserve the Local Label List to start a new group.

3.46 End of Logical IF Pass 2 _

This is received from Pass 1 as a one-word item and follows the statement(s) controlled by a Logical IF statement. The indicator for this "statement" does not pass through Next Statement Gen (NEXTSTAT) but is detected by Logical IF Statement Gen (LOGIFSTG). Should this indicator show up when it is not expected (which it never does), it would filter through the Statement Gen BRU Table to a JRS to P2ABORT.

3.47 ENTRY Statement

3.47.1 ENTRY Statement ENTRYSTA _ Pass 1)

Turn off the Local Trigger unless we have come in through Entry Common 1 from finding a LOCAL quote. If no SUBROUTINE or FUNCTION has appeared, give an error message that will delete the statement. Then there are three steps to processing ENTRY: the ENTRY name, the dummies, and the finishing up. Each has some peculiar facets, since ENTRY is kind of a peculiar statment.

The name must be permissible to be a Subprogram Definition (see Permissibility, 2.8.1). Otherwise, give a diagnostic and bypass the rest of the work on the ENTRY name. Set up the permissibility of SPROGDEF and register both the trait SPROGDEF and SPROG. Then, if this was not declared LOCAL, register the EXTERNAL trait, which will be used in Pass 3 to output an external DEF. (See Start Binary Output in OUTBO, 2.28.1). If we are within a FUNCTION (as opposed to a SUBROUTINE), we have to make the type of the ENTRY name match the type of the FUNCTION name. This can result in an interesting situation. Suppose the statement

```
      FUNCTION ALPHA(X)
```

appears. ALPHA is allowed to appear in a Type statment following the FUNCTION statement (for example, COMPLEX ALPHA). Normally it is all right that we do not know the type of a function until sometime later, but if an ENTRY comes along, we have to establish a type for the FUNCTION so we can give it to the ENTRY also. So fetch a pointer to the function's result scalar (previously set up in FUNCPNTR by Function Statement) and use Register Type to give it an implicit type if it does not already have one. Then call Inherit Type to give the ENTRY name the same type, if possible. The only way it might not be possible would be if the ENTRY name had already been typed, as in

```
      FUNCTION ALPHA(X)

      REAL ALPHA

      INTEGER BETA

      ENTRY BETA(X)
```

and this is diagnosed as an error.

Now to process the dummies (if any). Note that the conditions under which a name may appear as an ENTRY dummy are somewhat more lenient than for dummies of internal subprograms. In particular, an ENTRY dummy may have appeared previously as a dummy (but not necessarily). We will pile up on the Dummy List a group of dummy pointers and then build either a Dummy Group Plex or a Multiple Dummy Group Plex from them (even if there are none).

If the dummy is an ordinary identifier, make sure that either it is permissible to be a dummy under the normal rules, or that it already is a dummy. If neither, give a fatal ID conflict error. Otherwise, set up the permissibility and register the trait of DUMMY.

If the dummy is a multiiple dummy, call Multiple Dummy Scan 1 to scan it and "open" a Multiple Dummy Group Plex (see PLO POP). There should not be any more dummies after a multiple dummy. If there are , we go scan them, but this will probably result in all kinds of ridiculous plexes getting built. This does not matter (I hope) because we have given a fatal diagnostic anyway, and all of this will be thrown away eventually. Note that there is some confusion caused by the similarity between a multiple dummy and an ordinary dummy enclosed in slashes (as allowed for compatibility).

If there are asterisk dummies, they have to be counted and for each, a special Asterisk Pointer is put on the Dummy List. See below what the count is used for.

After all the dummies have been piled up, build the appropriate plex with them and file it following the ENTRYNO and the ENTRY name. Then check for excess right parentheses and for all the things that might casue the statement to

fail. This includes delayed fail, under Logical IF, in BLOCK DATA, inside DO loop. Make sure this line prints even if no LS (see PRIFSUM, 2.3.6), since ENTRY is a line of special interest.

Now that we know the statement cannot fail, we can do two irreversible things. The first is normally done by one of the statement exits, but we have taken over most of those duties already: that is to signal "not a jump statemnt". The second is to process asterisks that may have appeared. This whole feature is pretty strange, and included only for compatibility, but to do it cleanly, we have to allow asterisks to show up in an ENTRY statement <u>following</u> the RETURN that references them. For example,

```
      FUNCTION ALPHA(A,*)
3     RETURN 2
      ENTRY BETA(A,*,*)
      GO TO 3
```

Here, the RETURN 2 is legal if the program was entered through BETA. But we would like Pass 2 to diagnose the statement if no ENTRY shows up with at least two asterisks. So for each FUNCTION or SUBROUTINE (including internal ones), we keep the maximum number of asterisks that have appeared on the bottom of the Asterisk List. After each ENTRY, we update this count if the number of asterisks encountered is greater than had previously appeared. The RETURN statement in Pass 2 thus finds the maximum count for each subprogram in order on the Asterisk List and can produce lovely diagnostics.

Exit through Entry Exit, which avoids much of the work we have already done.

Three items are transmitted to Pass 2:

1. ENTRYNO
2. ENTRY name (pointer to symbol table)
3. Dummy Group plex (either ordinary or multiple)

3.47.2 ENTRY Statement Gen ENTRYSTG Pass 2

ENTRY statements generate the same subprogram heading code as generated by SUBROUTINE and FUNCTION statements, and it is generated by the same subroutine, Sprog Head Gen (SPHEADGN, 3.78.3). ENTRYSTG does not alter the following items established by SUBSTAG:

(1) FUNCTRIG: it will still be set if and only if the subprogram is a F FUNCTION;

(2) RTSPGTRG: the ENTRY point will have re-entrancy checking code if and only if the original SUBROUTINE or FUNCTION code did (note that the same Real Time Temp will be assigned to the ENTRY point by RTTEMPMK);

(3) SPNAMEVP (Sprog Name Variable Pointer): intervening ENTRYs do not change the identity of the variable used to return FUNCTION values;

(4) ASTCOUNT: the maximum asterisk dummy count applies to the entire subprogram, including all its ENTRYs;

(5) SPLINKPT (Sprog Link Temp Pointer): the ENTRY receiving sequence must store its link address into the same Private Temp used by the SUBROUTINE or FUNCTION receiving sequence; also, the Asterisk Return Vector must be the same--SPLINKPT defines the locations of both of these entities (see 3.78.2).

ENTRY Statement Gen first generates a branch around the subprogram heading code, so that statements that "fall into" the ENTRY statement simply fall on through it. Then it calls SPHEADGN to generate the subprogram heading code (see 3.78.3), and finally gens a Define Label item for the branch-around label. If the DEBUG option was specified, the ENTRY point name is registered on the ENTRYLST.

3.48 EQUIVALENCE Statement EQUIVSTA (Pass 1 only)

Processes a number of equivalence groups. Each group should begin and end with parentheses, but the compiler will diagnose and supply missing parentheses when it can determine that they belong.

For each group, the first word placed on the Equivalence List (EQUIVLST) is the true current line number. This may not be the same as the value LINECNT, which does not reflect continuation cards. This true line number is used in ALLOCATE in printing accurate allocation diagnostics. The line number has a special identifier, EQSETBIT, merged into it for ease in recognition by ALLOCATE.

Following this are pointers to each of the identifiers in the group. Each identifier that appears with subscripts is followed by a count of the subscripts and then the subscript values. These values are the actual 32-bit constants, not pointers. They are accumulated on the WORKLIST and then copied over to the EQUIVLST.

The following shows the eleven words that would be put on the EQUIVLST by the EQUIVALENCE statement processor for the statement

EQUIVALENCE (P,Q(3),R(-4)) , (X,Y)

1. EQSETBIT + current line number
2. Pointer to P
3. Pointer to Q
4. 1
5. 3
6. Pointer to R
7. 1
8. -4
9. EQSETBIT + current line number
10. Pointer to X
11. Pointer to Y

Note that at the time these values are placed on the EQUIVLST, it is not known (necessarily) whether the "subscripts" are actually subscripts or are position counts, or whether there are a legal number of them. This is determined later on during allocation (see ALLOCATE, Section 3.8).

If a name is not permitted in EQUIVALENCE (see Permissibility, Section 2.8.1), we put out a diagnostic, discard the name, and bypass any subscripts that may appear.

Exit through Declarative Exit 2 (valid in BLOCK DATA).

No code is transmitted to Pass 2, only the EQUIVLST data for ALLOCATE. Each equivalence group results in:

1. Beginning of group indicator (line number merged with EQSETBIT)
2. For each identifier in the group:
   A. Pointer to the identifier
   B. Count of subscripts (never zero; B and C absent if no subscripts)

C. The subscript values (full-word signed integers, not pointers)

3.49 EXTERNAL Statement EXTERNST _ Pass 1 only)

Actual operation is quite simple. Assuming each name is permissible in EXTERNAL (see Permissiblity, Section 2.8.1), it has new permissibility set up and has two traits registered, External and Subprogram. Then it exits through Declarative Exit 1 (invalid in BLOCK DATA).

The theory behind EXTERNAL, however, is a little complicated because of the various ways it can function:

1. The Subprogram trait assures that the name is classified as a subprogram (rather than a scalar) if it is never used as a subprogram (e.g., only passed as an argument -- see Name Alone Argument Scan, Section 3.10.1.3.5.2.2).

2. The EXTERNAL permissibility prohibits Intrinsic (see Process If Intrinsic, Section 3.10.1.3.5.3).

3. The EXTERNAL trait determines whether a SUBROUTINE, FUNCTION ENTRY, or Statement Function appearing in the program should be externally DEFed. (See "External Definitions", paragraph 5 of Section 2.29.2; "Output Function Groups", Section 2.30.5; and "Start Binary Output", Section 2.28.1.)

No code is transmitted to Pass 2, only changes in the Symbol Table.

3.50 FORMAT Statement

3.50.1 FORMAT Statement FORMATST _ Pass 1)

Most of the real work in processing FORMAT statements is done at run-time by the library routine 9IEDIT/9OEDIT. All Pass 1 does is to check for correct syntax, and then pack up the format specifications into a character string, eliminating superfluous blanks, and send it along. To do this, the Save Active Characters Flag (SVACTFLG) is set so that Next Active Character will save all active characters on the Alpha List, one character per word. Then, after the FORMAT has been scanned, we call Pack Alpha List, which packs the characters four per word and files them for Pass 2. Since the characters are taken off the Alpha List in the order they were put on, that is, top to bottom, we have to reserve the Alpha List before starting, so that it will not freight train (see Freight Trains, 1.4.10).

Format List Scan (below) is called recursively each time a nested left parenthesis is found. It keeps track of the parenthesis nesting level and

outputs a diagnostic warning if it exceeds ten. It goes ahead and generates the parenthesis anyway, because the run-time also checks for this. After scanning the statement and packing up the characters, we turn off the saving of alphanumeric characters. Then we have to do some of the "statement exit" work, since this is a rather special statement. After checking for delayed fail, logical IF, and BLOCK DATA, we can call NOLABEL to make sure that the statement has a label, since an unlabeled FORMAT statement is not too useful. Then exit through FORMAT Exit.

If N words of alphanumeric characters are present in the packed final version, then N+2 words are transmitted to Pass 2:

1	FORMATNO
2	Count of packed alphanumeric words following (i.e. N)
3 - (N+2)	Packed alphanumeric characters (four per word; last word filled out with blanks.

3.50.1.1 FORMAT List Scan FORLSTSC

Assumes a left parenthesis has been found. Checks the nesting level for greater than ten. Then if there is a right parenthesis immediately, it just exits. Otherwise, it calls FORMAT Element Scan continously until it finds a right parenthesis; then it exits.

3.50.1.2 FORMAT Element Scan FORMELSC

Only scans for correct syntax of a single FORMAT element (or specification). Does not do anything with it; Next Active Character is taking care of piling all the active characters on the Alpha List. If it does not find anything it recognizes, it puts out an error message and fails. The following points are of interest in this code:

1. Integer Scan, which is used to scan the various counts, is unlike Constant Scan, in that it will only accept an integer, will not ever convert it to floating (i.e., if it is too big), and does not register the value or return a pointer. It puts the 32-bit value in Integer Central. Most of the time, this value is subsequently ignored; however, the H specification needs it to count alphanumeric characters.

2. There are several places that do a CSA and then ignore the answer. This simply allows there to be a character, but does not care whether there was. For example, FN.3 and F.3 are both legal, and we do not care whether there was an N or not, but we have to step past it.

3. When an H, quote, or dollar sign alphanumeric string begins, we save the beginning character position, so that if it is accidentally too long (or unclosed entirely), we can flag the error at the beginning, rather than at

the end of the line. This makes the offending specification much easier to find.

4. When a left parenthesis is encountered, we go back to FORMAT List Scan, which will recurse back to FORMAT Element Scan.

5. Next Hollerith Character (below) is used by H, quote, and dollar specifications to pick up alphanumeric characters including blanks.

6. It is interesting that the compiler almost does not need to know about consecutive quotes (or dollars) being treated as a single quote. In a case like ('AB''CD'), the compiler mistakenly thinks that this is two 2-character quotes adjacent to one another, which is not what it is, but all the characters get sent out anyway and the run-time FORMAT scanner will process it correctly. The tricky case is ('AB' 'CD'), where the compiler thinks it can remove the blank, since it is not in either string. Doing this, however, converts the string to ('AB''CD'), which means something differenet. Our heuristic (i.e., easy) solution is simply to always include the next character after the quote (or dollar) on the Alpha List. If it is not a blank, it would get included anyway; if it is, it solves the above problem (without even being aware of the feature). Also, if it was a blank, we have to be sure to step past it (i.e. get it out of Current Character), since blank is not normally an active character and will not be recognized by FORMAT Element Scan when we come back in (which we will, because if the character is a blank, it is not a right parenthesis).

3.50.1.3 Next Hollerith Character NXTHOLCH

Used by places that have to scan alphanumeric fields, which include blanks as active characters. This routine steps past the current character (by zeroing CURRCHAR), sets the Use Blanks Trigger so that Next Active Character will consider blank an active character, and returns with a new character in CURRCHAR. If the character it comes to is an End Character (meaning end-of-line and no continuation), this is an error which we diagnose immediately and then fail. Rather than flagging the error way out here at the end of the line, we pick up the Hollerith Character Count that was saved previously and put the error marker there.

Originally, this routine went into use blanks mode, and then came out afterwards. However, with the addition of S in column 1 lines, which are scanned entirely in use blanks mode, it became necessary to save the current status of this mode and return to it when finished.

3.50.2 FORMAT Statement Gen FORMATSG _ Pass 2)

Code generated for the statement

```
        FORMAT (6X, 2F8.2)
```

is as follows:

```
        B       1G
        DATA '(6X,'
        DATA ' 2F8'
        DATA '.2) '
   1G   EQU     $
```

We begin by registering the source line number and current label on the appropriate lists (SRCLNLST and LABELLST) if the Debug option (DB) has been specified. We then create a pointer for the label (a plain label, as opposed to an active label; c.f., section 3.16) and generate a branch to it. All code currently on CODELIST is then assembled by ASMCODE; this clears CODELIST and insures that the FORMAT (which is to be copied directly to OUTLIST) will fall in the proper sequence. Now the word count is accessed from the Input List, and the specified number of words are copied from the Input List to the Output List. The word count is added to the location counter (LOCCOUNT) to account for the program space occupied by the FORMAT text. Finally, the branch-around label is defined, and FORMATSG exits to STEXNS (Statement Exit, No Script Gen) since it creates nothing which may involve "retroactive subscripts".

3.50.3 FORMAT Scan at Run-Time (IEDIT)

As described above, the compiler does not really "compile" FORMAT statements. It checks them for syntax errors, but the actually processing, as well as checking for further errors, depends on the values and input fields provided, and these are not known until run-time. This is analagous to the processing of the DATA statement, where an initial scan is performed on the variable list, but further processing and diagnosis is postponed until the constant values have been scanned. In this case, however, the constant values are present at compile-time, whereas in FORMAT processing they are not.

The compiler compresses FORMATs into text strings (eliminating unnecessary blanks) and includes these in the object program. For example:

```
        2    FORMAT ( F10.0 , I6 )
   2S            TEXT '(F10'
                 TEXT '.0,I'
                 TEXT '6)  '
```

```
READ(1,2) X, J

     LI,9 1

     LI,7 2S

     BAL,6     9BCDREAD

     BAL,6     9IODATA

     SNGL,C    X

     INTG,E    J
```

The FORMAT processor at run-time is called 9IEDIT (alternate entry point 9OEDIT). It is called via 9IODATA. These two routines interface with one another to provide output values and/or input fields for 9IEDIT to work on. 9IEDIT matches these values and fields with the FORMAT specifications, checks for further errors, and performs the necessary conversions.

3.51 FREQUENCE Statement (FREQSTA - Pass 1 only)

This statement does nothing, is provided only for compatibility with very old FORTRANs (e.g., IBM 704), and probably ought to be taken out entirely.

It will scan a number of groups, each consisting of a statement number (which we do register, so it will give a diagnostic if it is not defined somewhere) followed by a sequence of Integers enclosed in parentheses (which are discarded). Exit through Declarative Exit 1 (invalid in Block Data).

No information is passed to Pass 2.

3.52 FUNCTION Statement

3.52.1 FUNCTION Statement (FUNCTSTA - Pass 1)

There is a special entry point (FUNCOM1) for FUNCTIONs that begin with a LOCAL or type qualifier. Otherwose, we turn off the triggers indicating whether these were present. Then call File Return, which puts out a RETURN statement if necessary, to prevent people from falling into or branching to internal subprograms.

If there was no type specifically declared, we just let it go. It may have been declared previously (if this is an internal FUNCTION), it may get declared subsequently, or it may just acquire implicit type the first time the scalar name is actually used. In the latter two of these cases, we will find

out the type of the scalar and later have to inherit it back to the subprogram name (see below).

If there was a type declared, we want to impose it on the FUNCTION name, but first we have to check once again for one of s;IBM's;s curious "*n" data size appendages, which can be found dangling in the most unlikely places, such as after the FUNCTION name, when it obviously belongs after the type word. This refers to the following kind of statement:

```
REAL FUNCTION ALPHA*8 (X,Y,Z)
```

If we find an asterisk next, then we go through the Secondary Asterisk Scan Table (using a BAM, which is really acting as a JRS by mode) to get to the proper place to look for the *n's that are legal, given the type that was declared on this statement. Then we update the type pointer that we have been carrying in P0. In either case (asterisk or not) we call Set Type (in Expression Scan) to impose this type onto the FUNCTION name, and give a diagnostic if this is not possible.

Then, file FUNCTNO and MEF the SUBTRIG, so that FUNCOM3 (below), which processes both FUNCTIONS and SUBROUTINEs, will know which it has. If a left parenthesis is not next, give a diagnostic, but proceed; it is not very meaningful, but it will compile. Then go to FUNCTION Common 3 (below).

Four items are transmitted to Pass 2 (not counting the RETURN that may be created by File RETURN prior to the FUNCTION information):

1. FUNCTNO
2. Pointer to result scalar
3. Pointer to FUNCTION name
4. Dummy group plex pointer

3.52.1.1 FUNCTION Common 3 FUNCOM3 _

Used by both FUNCTIONs and SUBROUTINEs to handle both the subprogram definition and the dummy sequence. In the process, it does some very strange things, caused partly by internal subprograms and partly by the fact that the name of a FUNCTION is used twice: once as the entry point (or subprogram name) and once as the result scalar. Expects that the LOCALTRG and SUBTRIG have been set appropriately, and the subprogram name pointer is in WD.

Test for permissibility of this name to be a subprogram definition (SPROGDEF). If illegal, give fatal error, bypass the rest of this permissibility and trait testing, and go scan the rest of the statement anyway. Otherwise, set up the permissibility of SPROGDEF (see Permissibility, 2.8.1) and register both the SPROGDEF and SPROG traits. Note that SPROG is a class (or SOT) trait, establishing this name to be a subprogram (as opposed to scalar, array, etc.), whereas SPROGDEF is a flag (or SOF) trait, which only attaches an attribute to a name: in this case the attribute of having been defined. This attribute is tested in several places. Pass 2 uses it to decide whether to register the additional trait EXTREF (External Reference) or not (see Guarantee Addressable By Mode, which is used when a subprogram name stands alone as an argument, and Standard Sprog Call Gen, 3.28.4, used whenever a subprogram is called). Pass 3 uses it to decide how to list and output a reference to it (i.e., whether it is defined out there or in here: see Evaluate Pointer), in listing and outputting external definitions (see External Definitions, in the assembly PASS3F, and Start BO in the assembly OUTBO, 2.28.1), and in various categories of the summary listing.

If the subprogram was not specifically declared LOCAL, register the EXTERNAL trait. This trait is used in conjunction with SPROGDEF to ascertain whether a name is to have an external DEF. This trait may also have been registered by an EXTERNAL statement. In either case, if this subprogram does have the EXTERNAL trait, and it is an internal subprogram, and we are in real-time, give a warning, because internal subprograms are not reentrant and should not be called from outside.

Now comes some interesting logic. Consider the following sequence of statements:

```
X = F(Y)

. . .

FUNCTION F(D)

F = D

RETURN

SUBROUTINE GRUNCH
```

A = F(B)

Before and after the internal function F, F is considered a subprogram name. Within FUNCTION F, however, F is considered a scalar variable name. We cannot throw away the subprogram name when we reach the FUNCTION statement, not only because we are going to need it again in SUBROUTINE GRUNCH, but also because Passes 2 and 3 are going to need that name for generating the two calls (shown) to it, an external DEF, summary tables, etc. Still, we have to make sure that within FUNCTION F, the name is considered a scalar. So we call Create Identifier, which creates a new Symbol Table entry with the same name, and messes up the old one (by putting all ones into the first half of the name) so that it will not ever be found by a Symbol Table search. However, knowing that we are going to need that messed up name back again later, it remembers that name, and where it came from, on the Kludge List. You will see further on in this statement description how this is handled. (See also Finish Up, 2.9.4.) Now that we have this new identifier, we set up the permissibility and register the class trait of SCALAR. Then we put both the scalar and the subprogram name (a pointer to the now-messed-up but later-to-be-restored entry) on the Code List for Pass 2. In addition, we save a pointer to the scalar version of the name, so that if there are any ENTRY statements, they will know what scalar to return their results in. There is also a slight problem in doing that. We cannot just store that in FUNCPNTR quite yet, because if the statement should fail and be deleted, this was not one of the saved items, and it will reflect something that supposedly never happened. So we use SUBTRIG, which we no longer need, as a temp until we are sure the statement will not fail; then we can move it to FUNCPNTR.

Now we are ready to process the dummies. For each dummy, a pointer of some sort will get piled on the Dummy List, and then they will all be built into a dummy group plex. If there are no dummies, this plex will still be there but it will be empty.

If the dummy is an ordinary identifier, check for the permissibility of DUMMY (see Permissibility, 2.8.1) and set up the new permissibility and register the trait of DUMMY.

If the dummy is a multiple dummy, call Multiple Dummy Scan to scan it and "open" a Multiple Dummy Group Plex (see PLO POP). There should not be any more dummies after a multiple dummy. If there are, we go scan them anyway, which will probably produce garbage on the Plex List, but that is safe because it will all get deleted anyway by the error message. Note that there is some confusion caused by the similarity between a multiple dummy and an ordinary dummy enclosed in slashes (which IBM unfortunately allows).

If the dummy is an asterisk, a special Asterisk Pointer is used. Also, all the asterisks have to be counted, so that we know how big a number can appear on an IBM-type non-standard RETURN. This number may get bigger if succeeding ENTRY statements have more asterisks. In addition, if there are internal subprograms, each of them has its own count that must be remembered. So we remember them on the Asterisk List. Each FUNCTION or SUBROUTINE statement puts a new initial count on the Asterisk List, annt updates the appropriate

count (the current one on the bottom of the list) if it has more asterisks. Of course, like the FUNCPNTR (above), this cannot actually be done until we are sure the statement is not going to fail.

After all the dummies have been piled up, build either an ordinary Dummy Group Plex or a Multiple Dummy Group Plex out of them, and file it on the Code List. Then check for excess right parenthesis, and for all the things that might cause the statement to fail. This includes delayed fail, under Logical IF, in BLOCK DATA, and inside DO loop. Make sure this line prints even if no LS (see PRIFSUM,2.3.6) since FUNCTIONs and SUBROUTINEs are lines of special interest.

Now that we know the statement cannot fail, we can do some irreversible things: call the END LABELS Subroutine to close off the current batch of local labels (if there were any, meaning this is an internal subprogram). It also gives a diagnostic if any of the current local labels are undefined, because if they are not defined yet, they never will be. Turn off the Executable Statement Trigger, which says whether there have been any executable statements, and is used only by statement functions, which are not legal if there have been any executable statements, but are legal at the beginning of an internal subprogram (until another executable statement comes along). Turn on the SPROTRG, which is used by ENTRY and RETURN, which are only legal inside a subprogam. Store the pointer to the FUNCTION scalar (saved above in SUBTRIG) into FUNCPNTR where it belongs. Put the asterisk count on the Asterisk List. If there have been any statements at all (not counting comments), then this is an internal subprogram and does not affect the already made decision as to whether this is a main program or not. If this is the first statement, however, then this is definitely not a main program, so we turn off the Main Program Trigger.

One final kludge remains, as promised above. If this <u>is</u> an internal subprogram, and if there was a previous SUBROUTINE or FUNCTION <u>it</u> alos had its subprogram name messed up (by Create Identifier) in deference to its scalar name. Now that we are entering a new subprogram, we can call DEKLUDGE to put back the subprogram name (which was saved on te Kludge List.) However, there's also on the Kludge List the current subprogram name, which don't want restored until the <u>next</u> internal subprogram (or the end of the program). So we pull it off temporarily, de-kludge, and then put it back. Note that DEKLUDGE also inherits the type of the scalar back to the subprogram, if it does not already have one. For example, in the sequence:

```
      FUNCTION F(X)

      COMPLEX F

      F = CMPLX(X,-X)

      RETURN

      SUBROUTINE GRUNCH
```

the scalar F has acquired a type (Complex), but the subprogram name is still without one. Conversely, if the subprogram name had had a type (by being delcared before it appeared), then the scalar would have been given that type when it was created, and there will not be any conflict trying to inherit it back; nothing will be done.

Both FUNCTIONs and SUBROUTINEs exit through SPROGEX.

3.52.1.2 Multiple Dummy Scan MDMYSCN _ Check for the permissibility of Multiple Dummy and set up that permissibility and trait. Note that appearing as a multiple dummy disallows appearing as such again, except in ENTRY statements, whose multiple dummies can have appeared before. Put the symbol table pointer on the Dummy List, and scan the special scalar that represents the number of arguments (NARGS). Test and set up its permissibility. Note that the SUP word for NARGS differs from SCALAR only in not allowing it to be declared Logical, Complex, or Double Complex. Put the NARGS pointer on the Dummy List too. This is an extra word that is not exactly another dummy, but Sprog Head Gen in Pass 2 knows that when it processes it. Finally, give an error if there are any more dummies (multiple dummy has to be the last one), and open a Multiple Dummy Group Plex for whoever called.

3.52.2 FUNCTION Statement Gen FUNCSTAG _ Pass 2)

FUNCTION statements are handled just the same as SUBROUTINE statements in pass 2 with only one exception: Function Statement Gen sets the Function Trigger (FUNCTRIG) so that subsequent RETURN statements will cause a value to be returned. After setting the trigger, FUNCSTAG joins SUBSTAG (SUBROUTINE Statement Gen, see 3.78.2).

3.53 GLOBAL Statement GLOBALST _ Pass 1 only)

This is the only straightforward declaration statement in the compiler. It simply tests for the permissibility of GLOBAL on each name, and then sets up that permissibility and registers that trait. All the other declarations, even seemingly simple ones like EXTERNAL, INTEGER, ABNORMAL, NAMELIST, do something strange and/or additional. (Actually, GLOBAL also calls Allow Dimensions, so even this is not completely straightforward.) Note that the permissibility for GLOBAL allows GLOBAL, so if the name were declared more than once, this would be allowed, and the permissibility and trait would be set up again, redundantly.

From here on, Pass 1 is never interested in whether a name is GLOBAL. This fact is important only to ALLOCATE and to Pass 3, which has to output a DSECT for it (see STARTBO, 2.28.1) and list it properly in the summary.

No information is transmitted to Pass 2. Only the Symbol Table is changed.

3.54 GO TO Statement

3.54.1 GO TO Statement GOTOSTA _ Pass 1)

If the first character after "GOTO" is neither a digit nor a left parenthesis, go to Assigned GO TO Statement. If it is a left parenthesis, go to Computed GO TO Statement. Otherwise, call Active Statement Number Scan to get a statement number (and indicate that it is potentially branched to; this is used in deciding whether to remember index register contents after reaching a statement with a label). File a GOTONO and the pointer to the statement number. Exit through Jump Exit.

Two items are transmitted to Pass 2:

1. GOTONO
2. Statement number (pointer to either label list)

3.54.2 Ordinary GO TO Statement Gen GOTOSTAT _ Pass 2)

The ordinary GO TO statement is astonishingly easy to generate code for. We simply JRS to NXINWORD for the label and BIN a B instruction to it. Should the DEBUG option be specified, we precede the branch by a check-in call upon 908FHGO.

3.55 IF Statements

3.55.1 IF Statements IFSTAS _ Pass 1)

This is one of the horrible statements with multiple ambiguities in the syntax. A statement beginning:

```
IF (SENSE LIGHT 3
```

may be well be an IF SENSE LIGHT statement, but if it continues:

```
IF (SENSELIGHT3 * X +2.7
```

then it is probably an arithmetic IF statement, unless it continues:

```
IF (SENSELIGHT3*X+2.7 .GE. 5
```

in which case it is a logical IF statement. And unfortunately our top-down approach to expressions, and our insistence on punctilious scanning of even oddball statements (which prevents us from just decreeing that statements beginning "IF(SENSELIGHT" are IF SENSE LIGHT statements) forces us to do three separate scans over this, in order to find the one that works. Since arithmetic IF statements are probably the most common, we try that first, followed by logical IF, and then the FORTRAN II IF statements.

3.55.1.1 Arithmetic IF Statement Pass 1

Set the Fail Exit to Logical IF Statement. (See TRY-Fail, 2.7.) Scan for an Integer, Real, or Double Precision expression. Allow excess right parentheses and a comma afterwards. File ARIFNO and the expression pointer. Then scan and file three active labels (see Active Label Scan).

Now make sure the statement will not fail, so we can do something irreversible. There is a trigger called the After IF Trigger (AFTERIF) that is used in conjunction with the IF List to generate better code for arithmetic (and FORTRAN II) IF statements. It is helpful, when generating an IF statement, to know whether any of the labels it may branch to, happen to be the following statement, in which case you can fall into it instead. It is cumbersome in Pass 2 to try to go off and look at the next statement, however, when you are right in the middle of doing the IF statement. So, in Pass 1, we set the AFTERIF trigger at each IF statement that has labels (i.e. not Logical IF). Then, when the following statement comes along, we put its label on the IF List, so that Pass 2 can just take it off and look at it. If the following statement does not have a label, this is probably an error, since you cannot get there, but we put zero on the IF List instead, so that Pass 2 knows it cannot fall into it. However, if the following statement is a declarative statement without a label, it generates no code and is transparent to control falling through it. So we do not put anything on the IF List, we leave the After IF Trigger on, and let the first executable statement that comes along be the one whose label goes to Pass 2. Of course, if a declarative statement does have a label, then we go ahead and put it on.

This is all handled by Check Label After IF, which is normally called at one of the statement exits. However, we cannot wait until then to do it when we are processing an IF statement. We have to set the AFTERIF trigger for this IF statement, but before we do, we have to check to see if the trigger is already on because the previous statement was also an arithemtic IF, which may want to fall into this one, just as this one can fall into the next statement.

Exit through Arithmetic IF Exit.

Five items are transmitted to Pass 2:

1. ARIFNO

2. Expression pointer

3. Less than zero label pointer

4. Equal to zero label pointer

5. Greater than zero label pointer

3.55.1.2 IF SENSE LIGHT Statement Pass 1

Control comes here if the quote scan for "(SENSESWITCH" did not succeed. At this point we loop specifically for a left parenthesis, and if it is not there, we go off to try IF OVERFLOW. Otherwise, we proceed exactly as with IF SENSE SWITCH, except that: the quote scan is for "SENSE LIGHT", with no left parenthesis; a sense light may be in the range 1-24; we file an IFSLITNO. Exit is through Arithmetic IF Exit.

Four items are transmitted to Pass 2:

1. IFSLITNO

2. Sense light number (pointer to expression)

3. "On" label pointer

4. "Off" label pointer

3.55.1.3 IF SENSE SWITCH Statement Pass 1

Control comes here only if arithmetic and logical IF have both failed. So we rescan the word "IF" and then look for "(SENSESWITCH". Why this quote has the left parenthesis in it and SENSE LIGHT does not, I do not remember; there does not seem to be any good reason for it. Then we file an IFSNSWNO, and scan for an expression. If the expression is just an integer constant (which it often is), we check it for being in the range 1-4. Then we file the expression, check for excess right parenthesis , and allow a comma afterwards.

Then scan and file two active labels, and proceed exactly as in arithmetic IF to make sure the statement will not fail and then call Check Label After IF. Exit through arithmetic IF Exit.

Four items are transmitted to Pass 2:

1. IFSNSWNO

2. Sense switch number (pointer to expression)

3. "On" label pointer

4. "Off" label pointer

3.55.1.4 IF FLOATING OVERFLOW Statement Pass 1

Control comes here if there is no left parenthesis after the word "IF". (Admittedly it gets here in a terribly roundabout manner, having already failed on two whole statements and one quote. This could be improved considerably.) By use of cascaded QSAs, it checks for "OVERFLOW" or "FLOATING OVERFLOW" or "ACCUMULATOR OVERFLOW" and throws in "FLOATING ACCUMULATOR OVERFLOW" free of charge. If at least one of these does not succeed, we can finally give up on this as an IF statement, go back and start all over trying it as an assignment statement. If at least one does succeed, file an IFOVFLNO, allow a comma after the quote, and proceed with processing two active labels and the finishing up details (Check Label After IF, etc.) exactly as in IF SENSE SWITCH and arithmetic IF. Exit through Arithmetic IF Exit.

Three items are transmitted to Pass 2:

1. IFOVFLNO
2. Overflow label pointer
3. No Overflow label pointer

3.55.1.5 Logical IF Statement LOGIFSTA Pass 1

The basic part of this statement is pretty simple. The complicated part results from the fact that other statements are being scanned within the logical IF. A single logical IF in Pass 1 sends two "interface statements" to Pass 2. The first is a logical IF (LOGIFNO) and the other is an End of Logical IF (ENDLIFNO). In between comes all the information produced by the statements that the logical IF controls.

Control comes here only if arithmetic IF has failed. We have to rescan the word "IF" and the left parenthesis. Then we get a logical expression, and allow excess right parenthesis and a comma afterwards. Then we file a LOGIFNO and the expression pointer. Then we call Compound Statement Scan, which scans whatever statement(s) are controlled by the Logical IF and outputs information to Pass 2. It returns with an answer whether it succeeded or whether it found a major error and failed. If it failed, we also fail at this level. In other words, if any statement in a compound statement controlled by a logical IF fails, the whole logical IF fails. Then we file an ENDLIFNO (so that Pass 2 will know where to generate the bypass branch to) and exit through Logical IF exit.

Along with the above, however, are some machinations to provide a proper environment for the statements that are controlled by the logical IF. First of all, the AFTERIF trigger has to be reset, so that the controlled statements

do not think that they follow an arithmetic IF. However, the old setting has to be remembered, because the whole logical IF may follow an arithmetic IF. (See Arithmetic IF Statement, 3.55.1.1, for a discussion of this trigger.) After we come back from the controlled statements, we can restore the AFTERIF trigger and find out if the logical IF follows an arithmetic IF (see Check Label After IF, 2.9.9.6). But we have to have remembered the new setting of the trigger too--that set up by the controlled statements-- because one of them may be an arithmetic IF. For example:

```
      IF (X) 1,2,3

2     IF (Y 3) REWIND 5; IF(Z) 4,5,6

4     PRINT 7, X,Y,Z
```

The REWIND statement has to know that there is no arithmetic IF that can fall into it. (It would think there was if we did not reset the AFTERIF trigger.) The logical IF has to know that the arithmetic IF(X) can fall into it. (It would not know if we reset the AFTERIF trigger for the REWIND statement and failed to put it back the way it was.) And the PRINT statement has to know that the arithmetic IF(Z) can fall into it. (It would not know if we put the AFTERIF back the way it was before starting the logical IF and did not save the fact that the arithmetic IF(Z) had set the trigger.) So before scanning the controlled statements, we save the trigger and set it FALSE. Then, we save the new value and restore the old one and call Check Label After IF. Then we put the new value back. Complicated, what?

Similarly, the controlled statements have to know that they do not follow an unconditional jump (which would produce a diagnostic that the statement could not be reached). So we have to set the JUMPTRIG FALSE. But we have to save it to see whether the logical IF follows an unconditional jump. However, we do not want to then save the new setting of the JUMPTRIG, because even if the last controlled statement was an unconditional jump, it is still to get to the next statement, because the "FALSE" case of the logical IF bypasses the controlled statements and falls into the next statement. So we automatically set the JUMPTRIG FALSE after a logical IF.

The debug Null Pointer also has to be saved, so that each of the controlled statements can have its own, without interfering with the Logical IF's. Then, after making sure the statement will not fail (BLOCK DATA, delayed fail, end of statement), we can do the above-mentioned testing for arithmetic IFs. Also, we have to turn off the DO Error Trigger (which states that it is an error for a DO loop to end on this kind of statement -- see Process Label, 2.9.6). This might have been turned on by one of the controlled statements, thinking that it is illegal for a DO loop to end on it, but the DO loop would not really end on it; it would end on the logical IF, which is all right.

A logical IF is sent to Pass 2 in two separate parts. The first consists of two items:

1. LOGIFNO

2. Logical expression pointer

The second comes after all the information sent by the controlled statements, and consists of only one item:

1. ENDLIFNO

3.55.2 IF Statements Pass 2)

Unlike Pass 1, Pass 2 does not group the IF statements together. The fact that they are syntactically similar is Pass 1's problem and has been taken care of already. The only reason for grouping the statements together in this writeup is to parallel the part (3.55.1) describing Pass 1's treatment of IF.

As far as Pass 2 is concerned, the five kinds of IF statements are five separate and distinct statements.

3.55.2.1 Arithmetic IF Statement Gen AIFSTAG _ Pass 2 _

The fun part of Arithmetic IF is a routine called Generate 3-Way Branch (GEN3BRAN). This tests the three labels for various mutual equality and generates the branching code accordingly. It actually considers four labels, where the label of the next (following) statement is the fourth label; this allows it to generate code that "falls into" the next statement. There are fifteen different cases of object code that can be generated.

In AIFSTAG we first get the interface data. This consists of an expression pointer and three labels from the INPUTLST and a fourth label (the label of the next statement) from the top of the IFLIST. We get the three explicit label pointers through a subroutine called Next IF Label (NXIFLABL) which not only gets the label via NXINWORD but checks to see if it is an assigned label. If it is an assigned label, we register it on the Assigned Label List (which will be empty at the beginning of the statement) and return a created label with which to reference the assigned label. The assigned labels are then generated after the statement proper (by Generate Assigned Labels, ASLBLGEN) as if they were separate Assigned GO TO statements, each with a created label. If the same assigned label is used more than once in an Arithmetic IF statement, it is generated only once.

Having taken care of the labels, we now call routine to generate the expression: GTCCXOWN (Grntee In Condition Codes Own Mode; Is Sign OK?). This guarantees that the condition codes will indicate the positivity, negativity, or zeroness of the expression, which is evaluated in its own mode. Chances are that the value will also be in the accumulator, by maybe not. We also allow the expression to be evaluated with the wrong sign. If the sign is wrong, GTCCXOWN comes back with a FALSE answer, and we exchange the contents of LABEL1 and LABEL3, temps containing the 0 and 0 labels.

The reason that we use Grntee In Condition Codes rather than Grntee In Accumulator is that certain operations (like integer multiplication) do not set condition codes correctly. Also, as far as the "Is Sign OK?" part of it is concerned, GTCCXOWN will refuse to get a wrong sign if the expression is integer and the SBIT option was specified on the control card. When this option is not specified, the compiler is allowed to produce anomalous results with the number X'80000000'; with the SBIT option, it must get it right.

There is also DEBUG. If the DEBUG option is specified, we generate a call on 9DBFHIF before generating the three-way branch.

3.55.2.2 IF SENSE LIGHT Statement Gen IFSLSTG _ Pass 2 _

This is a pretty easy statement to generate. We evaluate an integer expression into register 9 for the light number. Then we call the library routine 9IFSLITE which tests whether the "light" is on (few computers have had real lights since the IBM 704) and skips if it is not. This is followed by one or two unconditional branch instructions to the labels in the statement. (If the second label is the label of the next statement, as indicated by the top of the IFLIST, then the second branch is omitted. For consistency with Arithmetic IF, this label is stored in a temp called LABEL4.

There is a reason for using LABEL4 other than consistency. Either of the two labels may be an assigned label, and if so we must generate on or two assinged GO TOs after the IF SENSE LIGHT statement. In this case, we cannot just "fall into" the following statement because there is intervening code. Therefore, Next IF Label (which is also used by Arithmetic IF) stores the label of the first assigned GO TO for this statement into LABEL4. Therefore, for the statement

```
      IF (SENSE LIGHT 13) 23,ZAP
```

the code comes out

```
      LI,9 13
      BAL,6    9IFSLITE
      B        23S
1G    LI,7 ZAP
      BAL,6    9ASGOTO
```

The label 1G is never referenced because the code can just fall into it. If the statement were written

```
      IF (SENSE LIGHT 13) ZAP,23
```

```
    23   CONTINUE
```

the fact that 23 is the label of the next statement could not be used because the ZAP code would intervene:

```
         LI,9 13
         BAL,6     9IFSLITE
         B         1G
         B         23S
    1G   LI,7 ZAP
         BAL,6     9ASGOTO
    23S  EQU       $
```

For a description of the subroutine NXIFLABL, see Section 3.55.2.1.

3.55.2.3 IF SENSE SWITCH Statement Gen IFSWSTG _ Pass 2 _

All but two of the POPs in this routine are shared with IF SENSE LIGHT Statement Gen (IFSLSTG) which is documented in 3.55.2.2. The only difference between the two is that IF SENSE SWITHC Statement Gen calls 9IFSWICH instead of 9IFSLITE.

3.55.2.4 IF OVERFLOW Statement Gen IFOVFLSG _ Pass 2 _

All but two of the POPs in this routine are shared with IF SENSE LIGHT Statement Gen (FSLSTG) which is documented in 3.55.2.2. There are two differences between the two: IF OVERFLOW Statement Gen calls 9IFOVFL instead of 9IFSLITE, and there is no integer expression to be evaluated since there is no switch or light number.

3.55.2.5 Logical IF Statement Gen LOGIFSTG _ Pass 2 _

The Logical IF Statement is rather a nuisance to generate because of the special-casing that goes on. The special case is a logical IF controlling a GO TO statement. If the statement is not a GO TO, we generate code that branches if the logical value is FALSE. The branch goes around the statement(s) controlled by the IF. If we were to do this with GO TO, however, there would be a redundant branch, i.e., a branch around a branch. So for GO TO we generate code that branches if the logical value is TRUE, and this branch leads to the label of the GO TO. (Naturally, this applies only to ordinary GO TO; assigned and computed GO TOs are handled just like other non-GO TO statements.)

There is a rub in this procedure. The Xerox Extended FORTRAN IV language allows a logical IF to control a compound statement. Now suppose the compound statement begins with a GO TO. Then there is logically no way to reach the statements of the compound after the GO TO. For example,

```
IF (X .GT. Y) GO TO 3; X = Y; REWIND 4
```

If the logical expression is TRUE, we go to 3 and do not do the X = Y or the REWIND 4; if the logical expression is FALSE, we skip all three statements, and we still do not do X = Y or REWIND 4. Pass 1 will flag this with a warning that the statement X = Y has no label and cannot be reached. But this is not a fatal error (alas) and Pass 2 has to generate reasonable code for it. If we did not try to special-case GO TO, this case would not cause any trouble. But we do, so it does. The FALSE branch of the logical expression is tempted to fall into the X = Y statement. We must not allow this to happen. So if a compound statement controlled by a logical IF begins with a GO TO but has other statements, then we generate an unconditional branch around the other statements. This case is checked for by looking at the next thing after the GO TO statement. If it is not an end-of-logical-IF, we generate an extra branch. So the aboe statement generates the following code:

```
        LW,8 Y
        CW,8 X
        BL      3S
        B       1G
        LW,8 Y
        STW,8   X
        LI,9 4
        BAL,6   9REWIND
1G      EQU     S
```

LOGIFSTG begins with a JRS to NXINWORD to get the logical expression pointer. It then simplifies the expression with SIMPEXP (Simplify Logical Expression) which is discussed in 3.11.2. Then we test the DEBUG Trigger, and if it is set we do not special-case GO TO. If it is either not a GO TO or the DEBUG trigger is on, we create a label and put it on the Special Label List (the push-down to whose bottom label the logical expression generator generates its branch), then calls EXPJUMPF (Logical Expression Gen Jump If FALSE). This branch will go around the statement(s) controlled by the IF. At this point, if the DEBUG Trigger is on, we generate a call on 9DBFHIL. Then we retroactively generate any subscripts that may be associated with the statement so far and assemble the code. The reason for assembling the code is that, if the statements controlled by the IF have any subscripts, we do not

want these subscripts to filter up to the front of the IF statement; they must be evaluated after the logical expression has been tested, so that they will be evaluated conditionally.

At this point we still have the indicator for the next statement sitting in WO. We test it for equality with ENDLIFIN (End Logical IF Indicator). The first time through, it will not be equal to this because a Logical IF cannot control a null statement (see 3.64.1). If it is not equal we call Statement Gen From WO (STGNWO), then NXINWORD, then branch back to the test to see if we have an End of Logical IF. When we finally get one, we define the label on the bottom of the Special Label List and branch to Statement Exit No Script Gen (STEXNS).

Now we get back to our basic special case: GO TO in the case that the DEBUG Trigger is not set. Here we JRS to NXINWORD to get the label to be gone to, then move this onto the Special List for the benefit of the logical expression generator. Then we JRS to Logical Expression Gen Jump If TRUE (EXPJUMPT) and discard the special label. Now we hope that the next "statement" is an End Logical IF: we JRS to NXINWORD and, if it is an End Logical IF, we BRU to Statement Exit (STATEXIT). If not, we JRS to Label Maker Branch Gen (LBMKBGEN) which generates a B instruction and gives us the label in WO; we append this to the bottom of the Special Label List and jump to the place that retroactively generates the subscripts and assembles the code (two paragraphs ago).

3.56 IMPLICIT Statement IMPLCTST - Pass 1 only)

To understand the operation of the IMPLICIT statement, you should first read the description of the Character Translate Table in section 2.4. At the beginning of each compilation, in a routine called Initialize Character Translate Table, the mode and flag fields of all the letters are set up. The letters IJKLMN are given type Integer. The others, including $ (which IBM claims is a letter), are given type Real, unless the ADP trigger is set, in which case they are given Double Precision type. If no IMPLICIT statements are used, that will never be changed, and that is the mechanism for implementing normal FORTRAN implicit type rules. When an IMPLICIT statement appears, it causes us to change the mode field of the appropriate letters. We could let it go at that; most FORTRANs probably do. But we felt it desirable to extrapolate (beyond what IBM defined) some rules for obtaining useful diagnostics. In particular, the use of both X and Y below causes problems:

```
YOOHOO = 0

IMPLICIT INTEGER (X)

IMPLICIT COMPLEX (X-Y)
```

X has been previously declared Integer and we try to declare it Complex. Y has not been declared anything, but the knowledge that Y is implicitly Real has already been used to make YOOHOO Real, so it is too late to let Y be declared a Complex letter. We handle this with two flags, contained in each Character Translate Table word. These are the Type Declared Flag (TYPEDCLD) and the Type Used Flag (TYPEUSED). Every time we declare a letter to have a particular implicit type associated with it (i.e., with an IMPLICIT statement), we set the TYPEDCLD flag, which prevents it from ever being declared again. Furthermore, whenever an identifier acquires its type implicitly (by being used in an executable statement without having had any explicit type declared), the routine Register Type determines what this type should be and then sets the TYPEUSED flag on that letter, which likewise prevents it from being declared.

Note a couple of facts. First, it would not work to start out with the mode fields empty and then just give an error if an IMPLICIT statement tried to give a type to a letter that already had a type. The letters do have types initially -- the normal FORTRAN IJKLMN rule types -- and these can be overridden. Once a type is declared or used, however, (even though it may not change) it can no longer be overridden. Second, it would work to do both of these tests with the same flag. The only thing we gain by having two is better diagnostics. This seemed particularly necessary in a case like:

```
YOOHOO = 0

IMPLICIT COMPLEX (Y)
```

where a diagnostic about conflicting type declarations might be pretty confusing to the user, who only has one IMPLICIT statement and does not realize he can get into problems just by using a name beginning with Y.

There are three nested loops in the processing of the IMPLICIT statement. The outer one looks for new type words. The middle one looks for sets of letters to be given that type. And the inner one steps from the lower letter to the upper one, applying the type and looking for conflicts.

The type words are found by the subroutine Type Quote Scan, which takes care of all six types, plus IBM "n" appendages, plus Automatic Double Precision (if the IMPLICIT statement says REAL, but ADP is set, we change it to DOUBLE PRECISION).

If the second letter precedes the first in the alphabet (not allowed), we give a diagnostic and reverse them. If a second letter is not specified, we set it equal to the first letter, so the inner loop will operate only once.

The inner loop takes advantage of the fact that the letters collate correctly. They are all negative (when left-justified, as they are) and "A" is algebraically less than "Z". "$" is positive, which means it collates behind Z, which happens to be just what we want. It is true that the Character Translate Table is sparse; there are lots of unused character codes between the letters. And if we have to run from some letter around to dollar sign, we even go through some used characters (punctuation marks). But it does not matter; we can attach mode and flags to these characters without harmful side effects, and they can never actually appear in an IMPLICIT statement.

Note the problem caused by TRY-Fail. An IMPLICIT statement may fail partway through. If it does, we insist on throwing out all vestiges of it, and this includes letters that may have already been given type and flags. (This is probably over-kill. We are beginning to think the increase in speed and cleanliness of the code would more than offset the loss of cleanliness in knowing exactly what is going to happen when a statement is rejected. If we did this again, we would probably just let whatever happened happen.) To guarantee this, every change we make to the Character Translate Table is remembered on the SYTCHLST (Symbol Table Change List), just like changes to the Symbol Table. Then, if the statement fails, everything is restored the way it was.

3.57 In-line Assembly Code

3.57.1 In-line Assembly Code SINCOLM1 _ Pass 1)

When the S option has been specified, and a FORRAN source line has an "S" in column one, the line is interpreted as containing (in columns 7-72) one or more assembly-style instructions separated by semicolons; this information is scanned by code which is completely independent of the normal FORTRAN statement scanners in STASCAN and is described in 3.57.1.2. The label field (columns 1-6) is processed normally by Label Field Scan before SINCOLM1 is entered; see 2.9.5.

The Pass 1/Pass 2 interface code that will be produced by SINCOLM1 is already in the form required by Pass 3; it simply passes through Pass 2. Thus, what we output is the Pass 1/Pass 2 interface code "In-Line Assembly & Instruction" (which Pass 2 will strip off), followed by the appropriate Pass 2/Pass 3 interface code, which Pass 2 will send along to Pass 3 with only slight modification: a pseudo-pointer ID representing "Absolute Addend" is appended to the Addend value. The Pass 1/Pass 2 interface item thus consists of four words:

1. The "In-line Machine Code" indicator
2. The Addend value
3. An instruction word in the form of a Pass 2/Pass 3 interface item (see 2.21)
4. The address operand pointer that goes with the above

3.57.1.1 Op Code List

The Op Code List, OPCODLST, is a static list used by SINCOLM1 to determine how to scan the various parts of an instruction. For each instruction mnemonic understood by SINCOLM1 (such as 'LW', 'MTW', etc.) there is a four-word item on OPCODLST containing (a) its name (up to eight characters); (b) the op code, in the form of a prototype Pass 2/Pass 3 interface item (Table 2.21.1) in which the op code has been filled in, but not the indirect address, R, or X fields; and (c) a word (called the Op Class Word) containing bits which tell SINCOLM1 how to scan and interpret the fields. These "class" bits are as follows:

Bit no.	Some ops using it	Interpretation
0	LCI, SLS	Special absolute op; address of special handler is in bits 15-31 of Class word (see below)
1	LI, AI	Immediate op code, 20-bit addressing
2	XPSD, SIO, RD	R-field optional
3	LW, AW	R-field required if neither bit set, R-field is not allowed.
4	all but MTW,...	R-field value to be in range 0 to 15 (normal), rather than -8 to +7 (as for MTW, MTH, etc.)
5	all but SLS,LI,..	Indirect address bit optional (not allowed otherwise)
6	all but WAIT,NOP	Address required (optional otherwise)

7	MI, SLS	Address must be absolute
8	all but LI,...	Index field optional (not allowed otherwise, as on immediate instructions)

The specialized shift ops (SLS, SAS, SCD, ...) and Load Conditions/Floating Flags Im edi
are handled specially: bit 0 of the class word is set, and bits 15-31 contain the ad res
scanning routine to be used in place of the ordinary one (see 3.57.1.3.3).

3.57.1.2 Scanning In-line Assembly Code

During the scan, we keep USEBLTRG (Use Blanks Trigger) TRUE so that blanks will be considered to be active characters.

In the following paragraphs, we will use the term "End character" to mean either a semicolon or the end of the source line.

Op Code scan

We skip over to the first non-blank character and initialize the symbol buffer (Identifier Central) to blanks (via INITSYML). Now we scan the letters and digits of the instruction mnemonic, packing them (via PACKSYML) into the symbol buffer (only the first eight are retained). When we get to the first character which is not a letter or digit we stop and loop up the symbol in OPCODLST. If found, we copy its prototype Pass 2/Pass 3 interface word (containing the op code) to OPCODE and get its Class Word to WO (we keep it there so we can do SWAs to test the class bits). We also reset the Kill R-field Trigger (KILLRTRG) so that the R-field will be properly updated when the time comes (see R-field scan, below). If the instruction mnemonic is not found on OPCODLST, then we give an error diagnostic and treat it almost like a NOP instruction: we put the NOP prototype in OPCODE, set KILLRTRG, and FET a class word specifying all additional fields as being optional, in order to minimize additional redundant diagnostics.

R-field scan

Now we look at the character that terminated the mnemonic. If it is a blank or End, then there is no R-field and we go directly to the Address Scan, first giving a diagnostic if the "R-field Required" bit is set in the op class word. If the first character is a comma, then there is an R-field: if it is not allowed by the op (i.e., both its "R-field Required" and "R-field Optional" bits are reset), we give an error message, skip to the first blank (or End) and go to the Address Scan. If an R-field is permitted, we call SCANVAL to obtain its (constant) value, check its range (0 to 15, or -8 to +7, according to class bit 4), shift it to the correct position, and store it in the R-field

of OPCODE (if KILLRTRG is set, we do not do this store, as it would destroy the NOP we substituted for an unknown instruction mnemonic). Now we error if the terminating character is not a blank or End, skip to the first blank or End, and proceed with the Address scan.

If the first character we looked at (the one that terminated the Op Code Scan) was not blank, End, or comma, then we produce an error message and skip characters until we do encounter one of these, which we treat as above.

Address scan

We initialize the address parameters as follows: (a) OPERAND (the address pointer) is set to ABSZERO, a pseudo-pointer representing Absolute Zero; (b) ADDEND is set to zero; (c) RESOLUTN, the address resolution code, is set to "WA" (Word Address resolution). Now we skip to the first non-blank character to locate the address field. In so doing, we skip over at most fifteen blanks; if no address field appears within fifteen columns from the command field, we assume that an address does not exist and proceed with Comment scan, first giving a diagnostic if the "Address Required" class bit is set. If there is an address field, and its first character is an asterisk, then we either set OPCODE's indirect address bit or give an error message, depending upon the state of the "Indirect Address Optional" class bit. Now we check to see whether the instruction is one of the special ones (bit 0 set); if it is, we call the Special Address Scan subroutine (SCANSPAD, 3.57.1.3.3) to get the absolute address, which we ORS into OPCODE (when building a special instruction, OPCODE contains the full 32-bit absolute instruction instead of the Pass 2/Pass 3 interface word with the instruction prototype in its right half); we then proceed to the Index scan. If the instruction is normal, we call Scan Address (SCANADR, 3.57.1.3.2) to get the absolute or relocatable address value. If it is relocatable and the instruction's 'Absolute Address" bit is set, we signal an error. SCANADR has caused the address to be represented in terms of (a) an address operand pointer in OPERAND, (b) an addend value in ADDEND, and (c) a resolution code in RESOLUTN. It has also set HEXTRIG if and only if the address was a positive hexadecimal constant; in this case the constant's value is specifically allowed to extend to the left of the address field and overlap the other fields of the instruction. To do this, we take ADDEND (which contains the full 32-bit hexadecimal value) and split it into two parts: the rightmost 17 or 20 bits (depending on the "Immediate Addressing Instruction" class bit) is to be used as the addend, so we put it back in ADDEND; the leftmost 15 or 12 bits are isolated, shifted to the right halfword, and ORSed into OPCODE. If HEXTRIG was not set by SCANADR, then we simply test the ADDEND value to see if it is in the correct range: -X'1FFFF' to +X'1FFFF' for normal instructions, -X'80000' to +X'7FFFF' for immediate.

Index scan

This scan is much like the one for the R-field. We test the first character for blank or End; if it is, we go to the Comment scan. If it is a comma, then

we have an Index field; if that is not allowed (the "Index field Optional" class bit is not set), we report an error, skip to the first blank, and go to the Comment scan. If it is allowed, we call Scan Value (SCANVAL, 3.57.1.3.1) to get the absolute Index field value; we check its range (error if not 1 to 7), position it correctly, and store it in OPCODE's Index field. Finally, we check the Index field terminator; if it is other than a blank or End, we signal an error.

If the first character encountered in the Index scan was not a blank, End, or comma, then we give a diagnostic and skip characters until we do find one of these, which we treat as above.

Comment scan

This is a simple scan: all we do is skip characters until the End (i.e., semicolon or end-of-line) shows up.

Finish-up

Now we must generate the output; how this is done depends upon whether or not the instruction is a special absolute. If it <u>is</u>, we output the following four words:

1. CODENO (the "In-line Machine Code" indicator)
2. Zero for the Addend value
3. A "Full Word Absolute" indicator (for Pass 2/Pass 3 interface)
4. OPCODE, containing the 32-bit absolute instruction

If the instruction is <u>not</u> a special absolute, then its address may have included the use of one or more Address Resolution Functions (BA, HA, etc.). We obtain the address resolution code from RESOLUTN and check it for "Word Addressing"; if it is <u>not</u> WA, then we build an Address Function Plex (ADRFPLEX) containing the RESOLUTN code and OPERAND pointer, and we store the plex pointer in OPERAND. Pass 3 understands this kind of plex; Pass 2 sends along the pertinent copy of the Plex List when it sees the "In-line Machine Code" indicator. SINCOLM1 generates the following item as output:

1. CODENO (the "In-line Machine Code" indicator)
2. The value in ADDEND
3. The Pass 2/Pass 3 interface word OPCODE containing most of the instruction
4. The address pointer in OPERAND

The last thing SINCOLM1 does before it exits is fix things so that Pass 2 will not think that this is an "executable statement" (and therefore generate a Statement Check-in for DEBUG mode), even though this is an executable statement for Pass 1's purposes. There is a "Null Statment" indicator on the Code List, and it is pointed to by the pointer in NULLPNTR. STASCAN is going to store an "Executable Statement" indicator in the location pointed to by NULLPNTR. Ordinarily, that would change the "Null" indicator to "Executable", but SINCOLM1 thwarts that by changing NULLPNTR to a harmless absolute address (the A register, in fact).

3.57.1.3 Address Scan Subroutine

3.57.1.3.1 Scan Value SCANVAL)

This subroutine is called to scan and pack the absolute values needed for the R-field, Index field, and special address fields. Part of it is also used by SCANADR for scanning absolute hexadecimal addresses. SCANVAL allows optional '+' or '-' sign, followed by a string of decimal digits, or a hexadecimal value in the form X'...' (where the elipsis represents a string, no longer than eight, of hexadecimal digits). It reports errors in syntax, and overflows. In the case of a positive hexadecimal value, SCANVAL sets HEXTRIG for the purposes of the Address scan (see 3.57.1.2).

3.57.1.3.2 Scan Address SCANADR)

This is the subroutine called to scan the absolute or relocatable address of a normal instruction. It represents the address in OPERAND, ADDEND, and RESOLUTN as described above (3.57.1.2) and tells the caller (via the Answer Flag) whether the address is absolute or relocatable. HEXTRIG is set if and only if the address is a positive hexadecimal value. The list of syntactic structures handled by SCANADR is quite a bit longer than that for SCANVAL:

*(1) $ (current location)

(2) = constant (literal)

*(3) FORTRAN identifier, like ABC6

*(4) special identifier, like $SIN or M:WQ

*(5) FORTRAN label, like 123$ or 995$

(6) decimal constant, like 995 or +2 or -76549

(7) hexadecimal constant, like X'5A' or -X'7FF'

*(8) address function, like BA(BUF) or HA(A)

In addition, the starred constructs may be followed by a positive or negative decimal addend value.

The first two items are recognizable by their leading character. If it is 'S', we create a label, define it at the current program location, and use its pointer for OPERAND The optional addend is scanned at SCADDEND, to be described later. If the first character was '=', then we set flags to allow strings (unless the '=' is followed by a '+' or '-') and not allow long hexadecimal constants (like 12Z...), and we scan for the optional leading '+' or '-'. Then we call GETCONST to scan the constant (unless it starts with 'X', in which case we call SCANVAL to scan the alternate hexadecimal form X'...'). We negate the constant if necessary, store its pointer in OPERAND, set ADDEND to zero, set RESOLUTN to "Word Addressing", reset HEXTRIG, and exit.

If the first character of the address is a letter, things are not so simple; there are four of the listed items which may begin with a letter: (3), (4), (7), and (8); we will not know until later which it is. We pretend we know it is going to be an identifier of some kind and start packing it into IDCENT. As long as we continue to get letters or digits, we keep packing the symbol (only eight characters will be retained, any more will be discarded). If we get one of the special identifier characters ':', '#', '@', or '_', then we know we have a special identifier and we continue the scan at SPECID, described below. If, instead, we get a quote character, and the symbol we packed turns out to be 'X', then we call SCANABSX to get the rest of the hexadecimal constant. If the termination character is '(' and the symbol was 'BA', 'HA', 'WA', or 'DA', then we have an address function; in this case we call SCANADR recursively to scan the address function's argument, set up RESOLUTN and shift the ADDEND value appropriately, and go to ADDEND1 to scan the optional addend. If the identifier scan was terminated in any other manner, then we have packed an actual FORTRAN identifier, and we arrive at FORTID where we register it on the Symbol Table, append its type (implicit) and class (scalar) if it is a new symbol, and generate a diagnostic (substituting an "absolute zero" pseudo-pointer) if it is INTRNSIC. From here, we proceed to SCADDEND to scan the addend, if there is one.

If the first address character is a special identifier character (':', '#', '@', or '_'), or if we encounter such a character during the scan of a FORTRAN identifier or decimal constant, then we arrive at SPECID which continues to pack letters, digits, and special identifier characters until something else comes along. We then have a special identifier in IDCENT; only the first eight characters have been retained therein -- if there were more than eight characters, we now give an error message (these identifiers are not supposed to be too long because of the ambiguity arising with a symbol like "ABCDEFGH#Q" which, when truncated to eight characters, looks like a FORTRAN identifier). We register the special identifer on the External REF List (REFLIST) and go to SCADDEND.

Another difficulty arises when the first character of the address is a digit: we could be getting any of items (6), (5), or (4) -- decimal constant, label,

or special identifier. As long as we continue getting a string of decimal digits, we do not know whether we will ultimately want the decimally packed value thereof, or these same characters packed as part of a special identifier. So we pack them both ways until a non-digit comes along. If the terminating character is a special identifier character, then we discard the decimal value and continue the special identifer scan at SPECID, as described above. If the termination character was 'S' or 'S', then we still have to resolve between label and special identifier. In this event, we go ahead and pack the 'S' or 'S' into the symbol, just in case. Then we look at the next character. If it is a letter, digit, or special identifier character, then the address is a special identifier and we continue its scan at SPECID. Otherwise, we have a local or non-local label number which is registered by RGLCLREF or RGNLCREF.

All of the above described scanners, except those for constants and literals, arrive at SCADDEND (or ADDEND1) to scan the optional deciaml addend, which, if present, is introduced by a '+' or '-' operator. The value is scanned by SCANVAL and added to ADDEND.

3.57.1.3.3 Scan Special Address SCANSPAD

When the instruction is one of the specially addressed absolute type (e.g., LCI, SLD), then bits 15-31 of its class word contains the address of a special address scan routine. The main Address scanner (see 3.57.1.2) calls SCANSPAD which, in turn, branches to the addressed routine. There is one for each of LCI, LFI, and LCFI, and one for all of the shift instructions.

The scanners for LCI and LFI demand an absolute address in the range 0 to 15; if the value is out of range, a diagnostic is given and the value is masked off to four bits. LFI uses the value as is, while LCI shifts it over four bits to the left. The LCFI scanner is prepared to allow either one value (in the range 0 to X'FF') or two (each in the range 0 to 15) separated by a comma. Thus, LCFI X'67' and LCFI 6,7 are equivalent.

The scanner for the shift instructions demands a value in the range -64 to +63, and masks it off to seven bits.

3.57.2 In-line Machine Code Statement Gen CODESTAG _ Pass 2)

If the Debug option (DB) has been specified, we register the source line number and current label (c.f., 3.14). We then pass along a copy of the Plex List to Pass 3, in case the address involves plexes, as it would in the following statement:

```
S     LI,9 BA(BUF)+3
```

Then the instruction itself is sent in three pieces: addend, op code, and address pointer. Finally, all index and DO-register contents are clobbered

(by CLOBALL) so that the compiler will not assume that possibly clobbered register contents are intact.

3.58 INPUT Statement

3.58.1 INPUT Statement INPUTSTA _ Pass 1)

This can be either an INPUT with list or an INPUT without list (i.e., using NAMELIST). Pass 2 must process those quite differently, but Pass 1 can process them both at once.

If there is a left parenthesis following the word "INPUT", go scan an integer expression and a right parenthesis. Otherwise, create a pointer to the card reader input unit number (105). Then, if there is anything more following (i.e., an I/O list), file the INPUTLNO (INPUT with List); otherwise file the INPUTNO (INPUT without list). Then file the unit number pointer. If there was a list following, go to Input List Scan (INLIST); otherwise exit through Active Exit.

There are two items transmitted to Pass 2 (not including the I/O items that will follow on an INPUT with list):

1. INPUTLNO or INPUTNO
2. Unit number (pointer to expression or constant)

3.58.2 INPUT With List Gen INPUTLSG _ Pass 2

INPUT With List is easier to generate than INPUT Without List (3.58.3) even though it contains an I/O list. The I/O list is handled as if it were a number of separate statements, so the only care we have to take for the I/O list is to RSV the DCODLIST. (The reason for doing this is explained in 3.61.2.3.)

We JRS to NXINWORD to get the expression pointer for the Unit Number and call CHEKUNIT to see if it is 105. (CHEKUNIT does a few other things that are explained in 2.23.) If it is not 105, we evaluate the expression into Register 9 and build a call to 9INPUTU; if it is 105, we do nothing about Register 9 and build a call to 9INPUTL.

3.58.3 INPUT Without List Gen INPUTSTG _ Pass 2)

We JRS to NXINWORD to get the unit number pointer. We then QJS to CHEKUNIT, having EAXed zero to indicate that this is a Read unit and not a Write unit.

CHEKUNIT tests the unit number for being constant and returns an answer TRUE if it is 105. We do not really care if it was 105 or not, and since CHEKUNIT discards WO if it is 105, we have to go to some extra care: we make an extra copy of WO before calling CHEKUNIT and then, if CHEKUNIT says FALSE, we throw it away.

The we see if the NAMELST is empty. If not (the only sensible case), there is a label pointer in NAMLSLOC, and we fetch that and generate a LI,7 instruction to it. Otherwise, we generate a LI,7 0 instruction, which at execution time will have the effect of an empty NAMELIST--i.e., no name will be found to be in the list. After that we BLC the SINPUT Link (i.e., generate a BAL,6 instruction) and BRU to Statement Exit (STATEXIT).

3.59 INTEGER Statement INTGERST _ Pass 1 only)

See Type Statements, Section 3.80.

3.60 Input/Output List

3.60.1 I/O List Scan IOLIST _ Pass 1)

The scanning of I/O lists is quite complicated, both because of the syntactic ambiguities involved and because it has to work both for normal I/O statements and for the OUTPUT statement, which is a very strange statement. At the upper level, it is fairly straightforward. There are several entry points that set triggers indicating whether this is for an OUTPUT statement, and whether it is a pure input list, meaning that only variables can appear (not expressions). Note that ordinary BCD input statements can, in fact, have expressions in them, since these can be used to satisfy N specifications in the FORMAT statement. For example:

```
      READ 5, X, J+K, Y

5     FORMAT (F10.3, F10.N)
```

However, the INPUT statement and the READ DISC and READ DRUM statements can only have variables in their lists. Actually the binary READ statements are the same, but their scanning was too bound with common code for the WRITE statements to make it worth while checking for this. The run-time will catch it anyway.

First, we reserve the Code List. This is because, if a DO control comes along, it only appears after all the code has been output to the Code List for the I/O items that the DO is supposed to control. It is necessary to get some code for "beginning of I/O DO" onto the Code List ahead of the I/O items, so

by reserving the Code List here at the beginning, we can copy away the I/O item code when a DO control comes along, file the DO control information, and then copy back the I/O item code.

In theory, we then repeatedly call I/O Item Scan, which generates the interface items for each I/O item in the list. Then, if there was no DO control, we can unreserve the Code List and we are ready to file an ENDIOLNO signaling the end of the I/O list. Here, however, the ambiguities of the DO control syntax get in the way. Look at the similarities between the following two I/O lists:

```
(A(I),I)

(A(I),I=I,5)
```

In the first case, I appears as a list item, in the second as a DO control variable, and with our "top down" approach to scanning, it is very difficult for us to tell which it is without a double scan (see Section 2.7.3). So, following the first I/O item, we have to TRY I/O DO Control first on every time. If it comes back FALSE, we can then go scan a regular I/O item. Otherwise we are done, and can fall into End Of I/O List (unless this is an OUTPUT statement, in which case FINOUT takes care of that. See Section 3.65.2.3.)

3.60.1.1 End Of I/O List ENDIOL

After all I/O list items have been scanned, and information transmitted to Pass 2 for them, an ENDIOLNO is filed to signal that there will not be any more I/O list items. If there were no items in the I/O list, this will get filed immediately after the beginning code for the I/O statement.

One item is transmitted to Pass 2:

1. ENDIOLNO

3.60.1.2 I/O DO-Control IODOCTRL _ Pass 1

Since this routine is always TRYed, we first have to save the lists that might be destroyed. This is the usual set for expressions plus the DO List. Then we scan for an Integer, Real, or Double Precision scalar and an equal sign. If these are not found, we will fail and return to the upper level with an answer FALSE; all the lists and the scan pointer will be returned to where they were, and the upper level can then scan for an ordinary I/O item. Once we have seen the equal sign, however, we might fail but we know for sure that this was intended to be a DO control and certainly will not succeed as an ordinary I/O item. So we set the automatic succeed flag (on the bottom of the TRY List) to prohibit a return with answer FALSE (see Section 2.7.4). Either the TRY succeeds (answer TRUE) or it fails at the upper level too.

Then strange as it may seem, before filing the beginning of DO code, we file an end of DO indicator (ENDDONO). This will follow the I/O items on the Code List, which has previously been reserved. Then we copy the Code List over to the DO List (for lack of a better place to put it), file an IODONO to indicate the beginnning of an I/O DO control, and call DO Control (section 3.38.1.1), which scans the three expressions, builds a triple plex from them, and files it. Now we have all the items Pass 2 needs at the beginning of the loop on the Code List, so we can copy back the rest of the items from the DO list and the loop code will all be in the proper order.

The beginning of an I/O DO Control transmits three items to Pass 2:

1. IODONO
2. Control Variable (pointer to Symbol Table)
3. Pointer to DO control triple plex, which contains the three expressions

The end of an I/O DO Control transmits one item to Pass 2:

1. ENDDONO

3.60.1.3 I/O Item IOITEM _ Pass 1 _

An I/O Item may be one of three things:

An array name (unsubscripted)

An expression of any mode (this includes a single variable or constant)

Another I/O list within parentheses

Note that an expression may also be contained within parentheses, so there is an ambiguity between it and an I/O list contained within parentheses. In this case, it makes no difference which we call it. However, there are other ambiguities that force us to do a multiple (TRY) scan. Consider the following three I/O items:

(X(K))

(X(K), K=1,10)

(X(K))*R

The first is an expression, but it will not hurt to scan it as a nested I/O list. The second must be a nested I/O list, and the third can only be an expression. Therefore, we try I/O list first. It will catch most of the nested expressions too, which is more efficient than the other way around. However, to guard against the third case above, not only must it find a valid nested I/O list, but it must also be followed by a comma, right parenthesis,

or end-of-line, in order to assure that it is not part of a further expression. (For further information on TRY-type multiple scanning, see Section 2.7.3.)

There is also a rather complicated interface with the processing required by the OUTPUT statement. These are mentioned below, but described more completely in Section 3.65.

An I/O item transmits two things to Pass 2:

1. IOITEMNO

2. Pointer to item. (This may be a Symbol Table pointer, or an expression plex pointer, or any of the various constituents of an expression, such as constant, subscripted array plex, function call plex, etc.)

The step by step processing of an I/O item is shown below.

1. (IOITEM) Is the first character a letter? If not, go to step 12 to test for left parenthesis.

2. First character is letter. Call OPENOUT in case this is an OUTPUT statement (see Section 3.65).

3. TRY ARYNAMSC (Array Name Scan) to see if this an unsubscripted array. If not, it must be an expression; go to step 7.

4. Unsubscripted array name. REG (Register) the IOLUSAFG trait on this identifier, so that, if it happens to be an adjustably dimensioned dummy array, Pass 2 will know that it needs to generate prologue code to compute the size of the array. It only needs to do this if the array has appeared unsubscripted in an I/O list. (See Section 1.13.1.)

5. Call CLOSEOUT. If this is an OUTPUT statement, some special processing will have to be done to handle unsubscripted arrays. See Section 3.65.

6. Otherwise (not OUTPUT statement) go to step 10.

7. (1882) Item must be an expression. Remember whether the first character was a letter or not, and call AMEXPSC (Any Mode EXPression SCan).

8. If the item began with a letter, and is addressable, then it is a variable and is permissible for either input or output. However, if it does not meet both of those conditions, then it is something else, such as a constant or a more complicated expression, and may not

appear in a strictly input list. In this case (if the INPUTRG is set), give an error message. This is only a warning, because the run-time will catch it too (by the protected bit on the calling sequence word).

9. In either case, call CLOSEOUT, in case this is an OUTPUT statement (see Section 3.65).

10. (1883) Got an item. File IOITEMNO and the single word pointer to the item, whatever it happens to be.

11. Exit.

12. (1881) First character not a letter. Is it a left parenthesis? If not, go to step 16.

13. First character is left parenthesis. Call FINOUT, in case this is an OUTPUT statement and this happens to be a nested list (see Section 3.65).

14. An item beginning with a left parenthesis may be either an expression or a nested I/O list. We try the latter first, since it can include the former as a subset. It must be a TRY, so that we can rescan it as an expression in cases like

    (X+Y)*A

    So TRY IOLISTSC (see Section 3.60.1.4 below) and if it succeeds, exit. There is no need to go through step 10 to file items, because this has all been taken care of recursively at the lower level.

15. Not a nested I/O list. Call RESETOUT so that, if this is an OUTPUT statement, the appropriate flags can be reset. (See Section 3.65.)

16. (1884) Not a nested I/O list and first character not a letter. This must be an expression (e.g. beginning with a digit, decimal point, etc.), unless this is an OUTPUT statement. If it is an OUTPUT statement, we call OPENOUT and then check for asterisk and quote, which are special items that can appear only in an OUTPUT statement I/O list.

17. Otherwise (not an OUTPUT statement), this must be an expression, so go to step 7 to scan for it. Note that the expression scan routine will fail if a valid expression is not there.

3.60.1.4 I/O List Scan within parentheses IOLISTSC

This works much like the top level of scanning for I/O lists, except that it is a TRYed routine, so a little special work needs to be done. The usual lists associated with expression scanning have to be saved, as well as the

UNITTEMP (used only by the OUTPUT statement). We scan past the left parenthesis to get at the I/O list within. The right parenthesis will be picked up by COMMAT4, the comma test. As described previously, we reserve the CODELIST, so that if there is a DO control, we can move everything off and insert the "beginning of DO" code. Then we recurse back and scan for I/O items, separately by commas. We look for a DO control after every item.. If we find one, we just exit, because there cannot be anything more (except that there must be a closing right parenthesis).

If we find the closing right parenthesis without any DO control (this is a rather unusual nested I/O list without a DO control), then we have to make sure that we have really found a complete I/O list and not just the initial subexpression of a longer expression, as in the case (X+Y)*R. We do this by assuring that there is a comma, end-of-line, or another right parenthesis following what we consider to be the closing right parenthesis of the nested list. If there is not we fail, the TRY comes back FALSE, and the upper level will then scan for an expression rather than a nested I/O list. Otherwise, everything is fine and we can unreserve the CODELIST and allow what we scanned to stand as a nested I/O list.

3.60.2 I/O List Generation (Pass 2)

There are three kinds of Pass 1/Pass 2 interface items for passing I/O lists:

1. "I/O Item", followed by a pointer to any one of the following:
   - a simple variable or array element
   - an expression (for output only)
   - an unsubscripted array
2. "I/O DO-Control", followed by the control variable and a REPEAT Triple Plex pointer
3. "End of I/O List"

3.60.2.1 I/O Item Gen IOITMGEN _ Pass 2 _

3.60.2.1.1 Theory

To explain what I/O ITEM GEN does, it is useful to explain what it used to do, and then compare that with what it does now. In the olden days (i.e., in the 9300 Compiler), I/O ITEM GEN generated a single call on either 9IODATUM or 9IOLUSA. 9IOLUSA exists both on the Sigma and the 9300 and is used to pass a

whole array (all the elements, in storage order) as I/O items: 9IOLUSA makes repeated calls on 9IODATUM to accomplish its job. 9IODATUM (which does not exist on the Sigma) passed one piece of I/O data--i.e., a scalar, array element, or Multiple Dummy element. On the 9300, the call consisted of a BMA instruction to 9IODATUM, followed by a word indicating the mode of the item and its address. This second word could be indexed and/or indirectly addressed--a great convenience for subscripts and dummies. Each item in an Input/Output list would produce one call on 9IODATUM. For example, the statement

```
READ (105,17) GLOP, GRUNCH, CH7, ABC, TV, FCC
```

would produce six separate calls on 9IODATUM, involving twelve words. In the Sigma FORTRAN, the 9IODATUM library routine has been dropped in favor of 9IODATA. The latter can pass more than one I/O item in a single call. The code for the above input/output list now comes out

Sigma Code		instead of	9300 Code	
BAL,6	9IODATA		BMA	9IODATUM
SNGL,C	GLOP		SNGL	GLOP
LOGL,C	GRUNCH		BMA	9IODATUM
CMPX,C	*CH7		LOGL	GRUNCH
INTG,C	ABC		BMA	9IODATUM
INTG,C	TV		CMPX	*CH7
INTG,E	FCC		BMA	9IODATUM
			INTG	ABC
			BMA	9IODATUM
			INTG	TV
			BMA	9IODATUM
			INTG	FCC
			BRM	9ENDIOL

Thus the Sigma code takes 7 words, versus 13 for the 9300. The R field of the words following the BAL to 9IDOATA contains bits for continuation (C) and end-of-I/O-list (E) as well as the usual protection Note: The protection bit (P) means "do not store into thi"; in I/O lists it is used on constants (where storing would clobber the constant) and expressions stored in temps (where clobbering will not hurt anything, but is almost certainly an error on the

part of the user). The abbreviations PC and PE are printed when the P bit is combined with C or E. Then, the continuation bit is used above on all words but the last, and the last word contains an E bit, which causes a call on 9ENDIOL (END of I/O List). The sequence

```
INTG,E      FCC
```

is equivalent to

```
INTG        FCC

BAL,6       9ENDIOL
```

The library routine 9ENDIOL is still present in the Sigma library, although it is seldom called in generated code. There are cases, however, where the E bit cannot be used and a call on 9ENDIOL must be generated. For example the statement

```
READ (105,17)  (A(I), I=1,10)
```

produces a call on 9IODATA that is enclosed in an implied DO loop. If an E bit were used in the above call, the I/O list would get ended ten times (once for each iteration) instead of just once. Therefore the compiler generates a call on 9ENDIOL outside the loop.

Using 9IODATA instead of 9IODATUM is nice from the standpoint of saving object code, but it requires much more cleverness on the compiler's part than was required on the 9300. On the Sigma, I/O ITEM GEN reads more than one "statement" from the input list in order to see if it can concatenate the calls. The logic of how it does its job can be understood by reading the PASS2F listing, but the rules are these.

1. If index bits are used, they can only appear on the first argument of a call on 9IODATA. Thus if several subscripted items appear,

   several separate calls on 9IODATA must be generated.) (Constant subscripts do not require indexing except in dummy arrays. For non-dummy arrays, constant subscripts can be handled with addends.)

2. Unsubscripted arrays cannot be handled by 9IODATA; each array requires a separate call on 9IOLUSA.

3. End of I/O list may be expressed with the E bit, provided there are no implied DO-loops in the way.

When the I/O item is an unsubscripted array (to be passed by 9IOLUSA) one additional piece of information needs to be passed: the number of elements in the array. The compiler stores this information as one of the items on the ARRAYLST. It is interesting to note that this is the only case this

information is ever used in generated code. The only other use of it at all is in the summary. The number of elements is passed to 9IOLUSA as a value in register 9. If the array has adjustable dimensions, the number of elements may be an expression rather than a constant. In this case the compiler goes through a few machinations. This expression needs to be evaluated only if the array name appears unsubscripted in an I/O list (a relatively uncommon thing). Therefore there is a trait called IOLUSAFG (I/O List Unsubscripted Array FlaG) which is registered BY Pass 1 if the array is so used. When Pass 2 generates the receiving sequence in which the adjustable array appears as a dummy, it tests the IOLUSAFG trait to see whether to generate code for the element count. If needed, the element count expression is evaluated just after the receiving sequence and the result stored in a private temp. This temp's pointer is then stored in the ARRAYLST to be used when the array comes up as an I/O item.

3.60.2.1.2 Details

Now that you know what calls on 9IODATA look like, I'll give details on how the compiler manages to generate them. It is surprisingly cumbersome to do, although the basic idea is simple: while we are processing the current I/O item, we need to know whether to append a C bit (Continutation) or an E bit (End of I/O List) to it; to do this, we read ahead on the INPUTLST and get the next "statement" and see what it is. If it is another I/O item that can be appended to the current 9IODATA call, we put a C bit on the current item. Then we generate the next "statement" without going through STATEXIT If the next statement is not something we can append to the current 9IODATA call, we do not generate the C bit. If the next statement is an End I/O List, we append an E bit to the current item, then suppress generation of a call on 9ENDIOL.

First we JRS to NXINWORD and check to see if we have an array (unsubscripted). If it is one, we generate code to load the number of elements into register 9. This will be a single instruction, either LW,9 or LI,9, and appears with the comment 'NUMBER OF ELEMENTS'. (The LW,9 is generated only for adjustable arrays.) Then we address the first element of the array (JRS ADR1STEL) and then QJS to I/O End Test (IOENDTST). This routine gets the next input word and checks to see if it is an End I/O List Indicator. If so, it discards the indicator from W0 and EATs an E bit, then exits TRUE. If the next input word is not an End I/O List Indicator, the indicator is retained in W0 and the routine EATs a zero constant and exits FALSE. Having returned from IOENDTST we temporarily ignore the answer it returned and we ADR a BLC instruction that generates a 9IOLUSA link. Obbiously, we are doing this ADR because we intend later to generate a

```
BAL,6    9IOLUSA
```

instruction. Why not, you may ask, simply ADR the 9IOLUSA link rather than ADRing a POP that reads

```
        BLC     L9IOLUSA
```

The reason is that sometimes we want to

```
        BLC     L9IOLUSA
```

and sometimes we want to

```
        BLC     L9IODATA
```

but sometimes we want to do nothing, in which case we ADR a

```
        NUL     0
```

POP. But to resume: we have ADRed our thing and now, with the answer flag still telling whether the next statement was an End I/O, we fall into some common code that is shared with the 9IODATA generator. (1) This common code, by the way, knows that it is about to generate the last item on the current call -- be it 9IOLUSA or 9IODAT. It may or may not append an E bit. First we JRS to IOARGGEN, then to SCRPTGEN, ASMCODE, and CLOBINDX (see 2.14.4, 2.15, and 2.17.5). (2) Now we are ready ot test the answer given by IOENDTST. If FALSE, we have two things on the bottom of the WORKLIST: W1 has the indicator that this was an I/O Item "statement" and W0 has the new indicator that did not turn out to be an End I/O. We do a

```
        STO     W1
```

thus discarding the old indicator, then BRU to Statement Gen From W0 (STGNW0) to handle the new statement.

Suppose tha answer flag is TRUE. In this case, IOARGGEN has already generated an E bit (with the help of the EATEMP) and we are almost done. We BRU to a place called ENDIOL1, which is the same as End I/O List Gen except that it omits generation of the call on 9ENDIOL. See End I/O List Gen, 3.60.2.3.

Now I have documented the 9IOLUSA case and have yet to document the 9IODATA case. But there remains the matter of the subroutine IOARGGEN. IOARGGEN generates an argument word for the I/O item, preceded

by a `BAL,6  9IOLUSA`

or by a `BAL,6  9IODATA`

or possibly, by nothing, according to the contents of P0: a BLC or NUL instruction ADRed outside. The argument word is a standard Calling Sequence arg word having the format described in 2.25.2.4. The C and E bits for this word are derived from EATEMP. If a subscript needs to be calculated, the code for this comes before the potential BAL instruction.

Now, consider the case where the I/O item was not an unsubscripted array. Therefore it is single item and will be passed by 9IODATA. (t may be a scalar, an array element, a multiple dummy element, a constant, or a non-trivial expression.) The first thing we do is guarantee it addressable in its own mode (GTADUOWN). This takes care of the non-trivial expressions. Note that we know this is the first item in a 9IODATA call, so the expression evaluation code will not interrupt the calling sequence. Then we ADR a

```
BLC       L9IODATA
```

POP and fall into some common code that is used by all but the last item in a 9IODATA call. We QJS to IOENDTST (I/O End Test) and if the answer comes back TRUE, branch to the previously mentioned common code for handling the last item of either a 9IODATA or a 9IOLUSA call. If this is not the end we check WO to see if this is an I/O Item Indicator. If it is not, then this is the last item for the current call and we branch to that same code for the last item. In this case, however, the answer flag will be FALSE so the E bit will not get generated. Suppose we did find another I/O Item Indicator. In this case, we are still undecided whether or not to terminate the current call; we have not yet generated the current item. So we JRS to NXINWORD thus getting the pointer for the next item and we start examining that pointer. First we check for addressability. If it fails that, then it cannot be the next item. We want to indicate this fact in the answer flag, with TRUE indicating that the next item cannot be appended. Since the

```
SOF       ADRSBLFG
```

has just come back FALSE, we branch to a place that sets the answer flag TRUE and then branch to the point just following the tests about to be described. We check for the other non-appendable items which are Constant Subscripted Dummy Array Plex, Fully Subscripted Array Plex (Dummy or not), and unsubscripted array. For any of these, the answer comes up TRUE.

At this point we have the next I/O argument in WO and the current one in W1, so we exchange them so we can work on the current one. We set up the C bit by doing

```
EAT,F     IOCBIT
```

(EATEMP was previously initialized to zero down inside IOENDTST.) Then we JRS to IOARGGEN. PO is set up to cause generation of a call on 9IODATA if and only if this is the first item in the list. Now again we interrogate the answer flag; if FALSE we

```
ADR       NULLOP
```

and branch to the common code. If TRUE (indicating no continuation), we JRS to SCRPTGEN ASMCODE, and CLOBINDX, and then BRU back almost to the very beginning of the I/O Arg Gen routine, for we have a brand new item in WO to generate.

3.60.2.2 I/O DO-Control Gen IODOGEN _ Pass 2 __

Implied DO-lists are handled like REPEAT FOR loops. The gen routine IODOGEN brings in the control variable pointer and the REPEAT Triple Plex pointer, and branches to Single REPEAT Item Gen (SREPGEN, described in 3.72.3). The latter routine generates the top-of-loop code and leaves the loop bottom code on the Delayed Code List. When the End-of-Loop item arrives, that code will be moved to the Code List. The End-of-Loop item is the same as for DO-loops and REPEAT loops; see 3.38.3 (ENDLOOPG).

3.60.2.3 End of I/O List Gen ENDIOLG _ Pass 2 _

End of I/O List Loop is handled at ENDIOL1 in PASS2F. We can get there in two ways:

1. Explicitly, via the Statement Gen BRU Table when the "End of I/O List" item is encountered at the statement level; or

2. Implicitly, via I/O Item Gen when the "End of I/O List" item is encountered in the look-ahead procedure performed by IOITMGEN (see 3.60.2.1.2).

When entered explicitly, we come in through ENDIOLG, which generates a BAL to 9ENDIOL. When entered implicitly, IOITMGEN has set the E (End of List) bit in the last I/O argument word it generated, so that an 9ENDIOL call need not be generated.

Each of the I/O statement gen routines leaves a file of end-code on the Delayed Code List which is to be copied to the Code List upon reaching the end of the I/O list. Usually, this file is empty. But for the ENCODE and DECODE statements, this final code is that required to store the "number of characters moved" value into the appropriate variable. ENDIOLG copies this file to the Code List, and, if it is not empty, calls CLOBALL to forget the contents of all X-regs and DO-regs (if the end-code is nonempty, heaven only knows how it will affect those registers).

3.61 LOCAL FUNCTION, SUBROUTINE, or ENTRY Statement (LOCALSTA) (Pass 1 only)

Set the Local Trigger (LOCALTRG) TRUE. Then if "SUBROUTINE" or "ENTRY" is found, go to Subroutine Common 1 (SUBCOM1) or Entry Common 1 (ENTCOM1). However, a Function may also have a type quote (e.g., REAL, INTEGER) between the word LOCAL and the word FUNCTION. So call Type Quote Scan (TYPEQS) to look for one. Ignore the answer whether it found one; if it did, it will have set the Type Trigger and put the appropriate type in P0. Now check for "FUNCTION", and if so, go to Function Common 1 (FUNCOM1). Otherwise, fail to Illegal Syntax (ILLEGSYN).

3.62 LOGICAL Statement (LOGICSTA - Pass 1 only)

See Type Statements, Section 3.80.

3.63 NAMELIST Statement

3.63.1 NAMELIST Statement (NAMELSTA - Pass 1)

If "NAMELIST" stands alone, with no identifiers, this means that all appropriate identifiers are to be NAMELISTed. This is done at the end of Pass 1 in Finish Up (see 2.9.4). So make sure the statement will not fail; then set the NAMELIST Trigger.

Otherwise, for each name scanned, call List Name On Name List (below), which determines if the name is already there, and if not, it puts it on, unless the name is not permissible to be NAMELISTed, in which case it says so, and we output a diagnostic and ignore the name. In any case, we allow it to be DIMENSIONed (see Allow Dimensions, 3.37). Exit through Declarative Exit 1 (invalid in BLOCK DATA), unless it was "NAMELIST" standing alone, in which case we have already done some of that testing, so exit through DATA Exit.

No items are transmitted to Pass 2. Only the Symbol Table and Name List may be changed.

3.63.2 List Name on Name List (LISTNAME)

Used by the NAMELIST statement, and by Finish Up (to list all appropriate names on the Name List). If the name is not permissible to be on the Name List, return with answer FALSE (see Permissibility, 2.8.1). Otherwise, set up new permissibility for NAMELIST. Then we want to search the Symbol Table to see if this name is already there or not. There is some complication in doing this. Consider the sequence of statements:

```
NAMELIST X

REAL X

NAMELIST X

X = 0

NAMELIST X
```

The first time X appears in NAMELIST, its Symbol Table pointer reflects UNUSED class and UNKNOWN type (see Identifier Scan). The second time, it has acquired REAL type. And the third time, it has also acquired SCALAR class. What we put on the Name List is a pointer to the appropriate entry in the Symbol Table. However, a pointer to the Symbol Table is not just a pointer to

the Symbol Table. These pointers are very special. Whenever a Symbol Table search is performed, the resultant pointer is always updated by picking up the class and type fields from the entry it is pointing to. Thus a symbol pointer does not say "Symbol Table" in its class field; it says UNUSED or SCALAR or whatever. Therefore, the three symbol pointers obtained in the above three NAMELIST statements would all have different bits set in the class and/or type fields, and they would not compare equal on a search of the Name List. There is a subroutine called Update Pointer that is often used to get around this problem. If you have an old symbol pointer (saved somewhere else earlier), and you want to make sure you know everything about that symbol that is currently known, you call Update Pointer, which goes into the Symbol Table entry and picks up the appropriate class and type and attaches them to your pointer. However, this technique will not do us any good here, because we would have to update every pointer we looked at on the Name List, and the SER POP is just not prepared to do that. So we do the opposite, so to speak. We declassify (or "downdate") each pointer before putting it on the Name List. This we do by masking out the class (or ID) and type (or mode) fields of the pointer and substituting the Symbol Table list number with zero type. Thus we have essentially regressed to "a pointer to the Symbol Table". This way, the same symbol will always have the same pointer. Admittedly, the above example is not something a user would be likely to write; this is provided more for the case where certain names are specifically NAMELISTed, and then ALL names get NAMELISTed (e.g., by being in DEBUG mode).

If the current symbol's pointer is not found on the Nmae List, then it is entered there.

3.63.3 NAME LIST Gen NAMELGEN _ Pass 2)

Pass 1 tells Pass 2 which names it wants included in the Name List by putting pointers to them on the list called NAMELST. To avoid confusion, we will henceforth call the former object, the table being generated by NAMELGEN during Pass 2, the Name Table. The names to be included thereon are determined by Pass 1 when it encounters a NAMELIST statement, or when, at the end of the program, it senses that the DEBUG option was specified, in which case it asks for the generation of a full Name Table.

NAMELGEN gens the table in two parts (1) a string of three-word items, each giving the eight character symbol name and the location associated with it; and (2) for each array in the table, a block of information describing the array structure. In general, the first part of the compiled Name Table looks like this:

```
nametabl  PZE      n                 number of names
          TEXT 'namel..."             1st name
          type locl                   location of 1st symbol
```

```
                    TEXT 'name2...'        2nd name
                    type loc2              location of the 2nd symbol
                    ...
                    TEXT 'namen...'        nth name
                    type locn              location of the nth symbol
```

where,

type is INTG, SNGL, etc., as in a calling sequence argument

loc is either the location of a scalar variable or label, or it is an indirect reference to an array description block in the second part of the table, which looks like this:

```
        block1      EQU     $              1st array description
block
                    PZE     m              number of dimensions
                    type loc-offset        offset address ("zeroth
                                                          element"
                                                          address)
                    type loc               address of 1st element
                    PZE     elcnt          number of elements
                    DATA mult1             1st subscript multiplier
                    DATA mult2             2nd subscript multiplier
                    ...
                    DATA multm             mth subscript multiplier

        block2      EQU     $              2nd array description
block
                    ...
                    ...
```

Each array description block has four fixed words, plus a variable number of words, one for each of the array's dimensions. These contain the corresponding subscript multipliers, calculated in words rather than elements.

NAMELGEN gens the first part directly in the Code List as it removes name pointers from NAMELST. Each time it encounters an array name, it creates a label to be used as the array description block address, builds the three-word item for the first part, RSVs the Code List, gens a Define Label item for the created label, generates the array description information, and moves this file of code to the Delayed Code List. Thus, the first part of the Name Table is accumulated on the Code List, and the second part on Delayed Code List. After all the names habe been genned, NAMELGEN combines the two parts of the generated Name Table on the Code List.

3.64 Null Statement two kinds)

"Null statement" has come to have two unrelated meanings. Originally it referred to a blank line in the source program. Rather than diagnose this as a syntax error, we allow it and call it a null statement. This kind of null statement is recognized only by Pass 1; nothing is sent on to Pass 2.

When the run-time debug features were added, each interface item from Pass 1 to Pass 2 was preceded by a special item indicating whether the following statement was executable and required check-in code to be generated or not. The indicator for executable type statements is EXECSTNO; the other is NULLSTNO and is referred to as a null statement indicator. This form of null statement is also recognized (and ignored) by Pass 2.

3.64.1 Null Statement Pass 1)

First the original meaning of null statement. This is really not a statement. It is a fancy name for a blank card. However, the compiler works on a "statement" basis, not a "card" basis, so what it finds is a blank statement, which may or may not be a blank card. To try and assure that it is, the following rules apply to a null statement:

1. It may not be embedded in a compound statement.
2. It may be the last part of a compound statement. (This can be thought of as a dangling semi colon on the previous statement.)
3. It may not have a label.
4. It may not be controlled by a Logical IF.

If these conditions are met, we bypass the statement exit procedures that apply to having found an actual statement. Nothing is sent to Pass 2.

Next the newer, run-time debug meaning of null statement. This kind too does not really do anything, since it is completely ignored by Pass 2. It only exists as part of the rather messy procedure that we have to go through to get the executable statement indicator to come out at the right places. It is a

little tricky to precede each statement with an indication of whether or not it is executable, since we do not know whether it is executable until after we have scanned it, by which time some interface data has already been generated. Probably this could have been handled by the DO-control method of moving things off the CODELIST, putting the indicator on, and then moving the other code back. But it was handled somewhat differently. At the beginning of statement scan (Section 2.9.2) we file a NULLSTNO indicator and remember where it is on the CODELIST (by keeping a pointer to it). Then, at statement exit, we reach back and change that NULLSTNO to an EXECSTNO if the statement was executable. If the DEBUG option is not requested, we neither file nor change this indicator. Furthermore, we ought not to change it if the statement is an END statement, because an executable RETURN/STOP has already been filed, and making the END executable results in two pieces of check-in code.

3.64.2 Null Statement Pass 2)

The null statement in Pass 2 is handled simply by its entry in the Statement Gen BRU Table (STGBRUTB) which is a BRU to Statement Exit No Subscripts or Assembly (STEXNSA). As pointed out in Section 3.64, the kind of null statement that this is dealing with is not the blank line in the source program, but the null statement sometimes generated by STASCAN under the DEBUG option.

3.65 OUTPUT Statement OUTPUTST _ Pass 1 only)

The OUTPUT statement is rather complicated and rather peculiar. One of the reasons it is peculiar is that its existence is known only to Pass 1. Pass 2 does not generate an OUTPUT statement, nor is there any special run-time routine to handle it. Pass 1 takes care of the whole thing by converting each OUTPUT statement to one or more pairs of WRITE and FORMAT statements. (If the unit number is 108, either explicitly or implicitly, Pass 2 converts the WRITE to a PRINT. For simplicity, we will use that unit in most of our examples.) For example, the OUTPUT statement below is converted into the two statements following it.

```
      OUTPUT X
-----------------------------
      PRINT 1G, X
1G    FORMAT(' X = '1PG.6)
```

The 1G is a generated label, which could not actually be written in a source program as shown above, but it is transmitted to Pass 2 exactly as if it had been written that way. Of course it may be 2G or 3G or whatever; these labels are created in sequential order.

The 1P is used so that values that come out in E-form will be in "scientific notation", i.e., with one digit before the decimal point.

Whenever possible, the OUTPUT statement will create one long PRINT statement and one FORMAT to handle additional list items. For example,

```
      OUTPUT X, K, A+B
-----------------------------
      PRINT 1G, X, K, A+B
1G    FORMAT(' X = '1PG.6/' K = 'G/' A+B = 'G.6)
```

Each new item has a FORMAT specification separated from the previous one by a slash, so that it will come out on a new line. Items of different types use slightly different specifications, though all use some form of G format. Since a P scale factor continues to apply once it has been declared, it is not necessary to keep putting out the 1P. So a trigger is kept to prevent it from being generated after the first time.

There are, however, two types of item that cannot be combined with other items in the same I/O list and FORMAT. The first is an unsubscripted array. Since we do not necessarily know how many elements it has, we cannot provide a FORMAT that has exactly the right number of specifications. We have to provide a parenthesized group that can be used over and over until the array

is exhausted. Such a parenthesized group has to be the last thing in the FORMAT statement, because that is how rescan of a FORMAT works. Thus an unsubscripted array can be tacked onto the end of an I/O list/FORMAT containing other items, but nothing can come after it.

A nested I/O list with a DO-control is even worse. We do not know how many times the loop will be executed and therefore how many items will be output. Under most simple circumstances, we could follow the same technique as with array names, namely use a parenthesized group at the end of the FORMAT statement. Unfortunately, there are cases where this does not work. Consider, for example,

```
OUTPUT X, ((MM(J,K), J=1,L), AA(K), K=1,M)
```

This would require two levels of parenthesized FORMAT groups, which is not possible. The "K" loop would be parenthesized and would be rescanned M times. It would contain the "MM(J,K) = " and the "AA(K) = ". However, the "J" loop is contained within the K loop, and it has to be rescanned L times, and there is no provision for multiple levels of automatic rescan in FORMAT statements. The only way to do this would be to control each of the parenthesized groups with an N. For example.

```
FORMAT( N( N( ' MM(J,K) = 'G/) , ' AA(K) = '1PG.6/) )
```

It gets extremely complicated to keep the slashes straight and to pass along the right values for N, especially when you consider that I/O loops are dynamic (like REPEAT loops), so there are cases where we cannot even know at run-time how many times the loop is going to go, much less at compile-time. What all of this means is that we cannot rely on automatic FORMAT rescan at all on nested I/O loops. (We could if they did not have DO-control, but most nested loops do have DO-control, and we do not know whether they do until we have scanned to the end of the nested item.) We have to move the DO-control to the outside of the PRINT statement and use a unique FORMAT statement for each level of nesting. Thus the rescan of the FORMAT results because we keep issuing PRINT statements that address the FORMAT statement. This is illustrated in the following example, along with the other kinds of OUTPUT list items.

```
      DIMENSION MM(12)

      OUTPUT X, X+SQRT(Y) , 'HELLO' , MM , Y , (MM(J),J=1,5) , N , *

      PRINT 1G, X, X+SQRT(Y), MM

1G    FORMAT(' X = '1PG.6/' X+SQRT(Y) = 'G.6/' HELLO'/' MM = 'G/(6X,G))

      PRINT 2G, Y

2G    FORMAT(' Y = '1PG.6)
```

```
      REPEAT 4G, FOR J=(1,5)
      PRINT 3G, MM(J)
3G    FORMAT(' MM(J) = 'G)
4G    CONTINUE
      PRINT 5G, N
5G    FORMAT(' N = 'G/' *END*')
```

Here the single OUTPUT statement has been broken up into four PRINT/FORMAT pairs, one of which is surrounded by a DO-control that used to be inside it. It becomes obvious that this conversion is not entirely trivial.

Some other comments about the above example: Alphanumeric strings (e.g., 'HELLO') and asterisks do not produce items in the final I/O list. They only produce alphanumeric fields in the appropriate FORMAT. Unsubscripted arrays produce an initial line with the name of the array. The parenthesized group that follows is used for all the succeeding elements of the array. It does not repeat the name of the array, but it does line up the values under the value on the initial line. It does this by counting the characters in the " name = " field and producing an X field of equal length in the parenthesized specification. Note that FORMAT rescan automatically produces a new output record. If the FORMAT is broken up, and there are floating point values in succeeding parts, a new 1P scale factor will have to be issued.

Most of the conversion work is done by a group of subroutines that are called at key points.

OUTPUT lists are processed by the same code that processes normal I/O lists (see Section 3.60.1). The I/O list generation is very similiar in the two. The creation of FORMAT statements and the breaking up into several WRITE statements are done primarily by some subroutine calls inserted at key points in the I/O list processing. Thus the actual code in the "OUTPUT Statement" section of the compiler is pretty small. Both are described in the subsections below.

3.65.1 Initialization

If a unit number is specified (as indicated by a left parenthesis after the word OUTPUT), we call Integer Expression Scan to scan it. If not, we substitute 108. Because we determine whether there is a unit number by whether there is a left parenthesis, this introduces the rather unpleasant restriction of having to follow the word OUTPUT by a comma if your first output item begins with a left parenthesis. But not much could have been done about this.

We allow a comma, either after OUTPUT or after the unit number. Then we have to make sure that there is something in Current Character (CURRCHAR). There may or may not be otherwise, because the CSA for comma may or may not have stepped past something. In order for the saving of alphanumeric characters to work correctly, we have to be consistent about whether there is something in CURRCHAR or not.

Suppose that the OUTPUT statement is going to have to be broken up into several WRITE statements. For each one, the unit number will have to be loaded. If the unit number is a constant, this is easy: we do a load immediate. But if it should be a complicated expression, we do not want to have to keep evaluating it. In fact, we really should not keep evaluating it, because it might contain a function call that should not be done more than once, or its constituents might change during execution of the I/O list (unusual but possible). In this case, we call PITEMPMK (Private Integer Temp Maker) to request a temp. We store the unit number in that, and from then on, if we have to generate a new WRITE statement, we will just load the temp. There are two tests necessary for a unit number to be "simple". It must be addressable, of course, But the usual definition of addressable allows a thing to have complicated subscripts and still be addressable. For Pass 2's purposes, this is appropriate. Here it is not. We do not want to have to recompute subscripts either. If the subscrips are all constant, fine. We can still access the value with a single load. But if it is a Fully Subscripted Array Plex, this is no longer simple.

Before branching to I/O List Scan (Section 3.60.1), we call RESETOUT (Section 3.65.2.3) to initialize the OUTPUT parameters.

3.65.2 Interface with I/O List Scan

As described above, there are two main things that have to be done specially for OUTPUT statements, namely, create FORMAT statements, and break up the I/O list into possibly several WRITE calls, each with its own FORMAT. This is done largely by three subroutines, OPENOUT, CLOSEOUT, and FINOUT. These are described in greater detail in following subsections, but speaking generally, we can say that OPENOUT is called just before each item, CLOSEOUT is called just after each item, and FINOUT (or its other entry point RESETOUT) is called whenever it is necessary to finish off one FORMAT statement and/or begin another. Thus it is called <u>initially</u>, as well as <u>after</u> every unsubscripted array, <u>before</u> every nested I/O list, and <u>after</u> every nested I/O list that contains a DO-control.

FORMATs are built, character by character, on the Alpha List. For each item, OPENOUT begins the FORMAT specification, the SVACTFLG (Save Active Characters Flag) is set so that scanning of the item will cause its characters to be saved on the Alpha List, and then CLOSEOUT tacks on the equal sign and the appropriate G format. FINOUT then attaches the final right parenthesis, generates an ENDIOL (End of I/O List), and then transmits the label of the FORMAT (which was created by OPENOUT) and combines the characters on the Alpha List four per word and sends it off as a FORMAT statement.

After every FINOUT (as well as initially), the FRSTSPEC trigger is set, indicating that a new WRITE/FORMAT pair must begin. When this is the case, OPENOUT does two things different. It begins the FORMAT with a left parenthesis instead of a spearating slash. And it generates a new WRITE, creates a label for the later to be generated FORMAT statement, and uses this label on the WRITE statement.

There is some additional processing that is not done by these subroutines. Following an unsubscripted array, we have to create the extra parenthesized group that will be used for all but the first element of the array. This consists of slash, left parenthesis, nX, the same G format that was used on the first line, and finally a right parenthesis. This of course is always followed by a FINOUT. The nX, as mentioned above, is so that the subsequent values will line up under the first line. The value of n is determined by remembering the number of characters placed on the Alpha List. This is done with ALPHACNT, which always indicates where the most recent item finished with the ALPHALST. Thus, if you put some characters on, you can count the ALPHALST and subtract ALPHACNT to see how many characters you put on. It is not always easy to know, since most of them appear automatically while scanning the item (due to the SVACTFLG).

No OPENOUT or CLOSEOUT is done before or after a nested I/O list, because it will do its own. However, the FINOUT must be done before. Also, if the TRY on IOLISTSC fails, we have to call RESETOUT to reset the various counters that may have been altered during the TRYed scan. Ordinarily it would be necessary to save these counters during a TRY, but since this TRY is always preceded by a FINOUT, we know exactly what they should be. (They should all be FALSE.)

Ordinary list items (variables, constants, expressions) require no special work, just an OPENOUT before and a CLOSEOUT after.

An asterisk (which can appear only in an OUTPUT statement I/O list) causes itself and the characters END* to be moved onto the Alpha List between the ususal quote marks.

When a quote sign is en ountered, we first remove it from the ALPHALST (because one has already been put there by OPENOUT). Then we repeatedly call NXTHOLCH (Next Hollerith Character), which scans characters in "Use Blanks" mode and, since the SVACTFLG is set, piles them onto the ALPHALST. When we find a second quote sign, it may or may not be the final one. If it in turn is followed by another quote sign, then it is a double quote (used within quote fields to represent a single quote), so we just let it go on the ALPHALST too and keep going. If it is only a single quote, we leave it on the ALPHALST, but not the following character. Also, if the following character is a blank, we get it out of CURRCHAR, because it is not normally an active character and it will confuse the scanning routines.

Also, on asterisk and quote items, we do not do a full CLOSEOUT, because we do not want the equal sign and the G format: we go to CLOSOUT1, which bypasses this part.

3.65.2.1 Open OUTPUT Item OPENOUT

This is called when the scanning section has determined that a real live I/O item is beginning. (A nested I/O list is not a real live item; it will contain real live items, but its opening left parenthesis does not belong on the ALPHALST.) If this is not an OUTPUT statement, it exits.

It turns on the SVACTFLG, so that the scanning will save all the active characters. Then it puts four characters onto the ALPHALST. Normally these are

/' x (slash, apostrophe, blank, and x)

where x is the current character (i.e., the first character of the item). It is necessary to put that first character on now, because it has already been scanned and Next Active Character will not see it again to save it.

However, if this is the first specification (item) in the FORMAT statement, the slash is a left parenthesis instead and we also have to start a new WRITE statement. So we file a WRBCDNO, followed by the unit number pointer, which has been saved in UNITTEMP. Now, if the unit number was a complicated expression, it was stored in a temp, and a pointer to that temp is in UNTMPTMP. So we pick up UNTMPTMP and store it in UNITTEMP so that, from now on, the temp will be used if any new WRITE statements have to be created.

Then we call MAKELBL to make a generated label and give us a pointer to it. We file this pointer on the CODELIST (for use by Pass 2 in generating the BCD WRITE statement) and we also save it so that it can be used on the FORMAT statement that FINOUT is later going to create.

3.65.2.2 Close OUTPUT Item CLOSEOUT

This is called when a real live I/O item has just finished being scanned. Like OPENOUT it exits (but FALSE) if this is not an OUTPUT statement. The (or whatever) to be output have been piled up on the ALPHALST behind what OPENOUT put there. There may also be one extra character that we do not want, which is the next character after the item. Most scans need to see the next character to be sure that their item is complete. However, some scans do not (such as L(1), where L is a logical array; the scanner knows that the right parenthesis is sort of final, and it looks for logical operators with a QSA that does not distrub anything, including CURRCHAR, if it fails). So, if there is anything in CURRCHAR, we lop off the last character from the ALPHALST.

Then we set up the XCOUNT, in case this is an unsubscripted array. We count the ALPHALST and subtract the previous count (kept in ALPHACNT), then subtract 3 to account for the slash (or left parenthesis) and the two quote signs that are on the ALPHALST but will not appear on the printed output.

Now we call FSBYMODE (FORMAT Spec By Mode, below) to generate the appropriate G format, based on the mode (type) of the item. We also use the type to make a stab at giving a warning message if the item's character string is so long that it and the value will not fit on the output line. We do not really know how big the output value will be, since it uses a widthless format, but we know the minimum it could be and if even that is too big, we give a diagnostic.

Finally (and this is done at the entry point CLOSOUT1, which is used by quote and asterisk items), we set up the new ALPHACNT and turn off the SVACTFLG (until it is turned on again by OPENOUT) and the FRSTSPEC trigger (so that OPENOUT will continue this WRITE/FORMAT pair rather than starting a new one). We exit TRUE to signal that this was an OUTPUT statement.

3.65.2.3 Finish OUTPUT Group FINOUT

This is called after each unsubscripted array, before each nested I/O list, and after each nested I/O list with a DO-control. (Actually it is called before any item that begins with a left parenthesis, which would include some expressions. These do not really need new WRITE/FORMAT pairs, but it is hard to prevent, since the syntax is very ambiguous and we do not know if there is going to be a DO-control until near the end of the parenthesized item.) Like CLOSEOUT it exits FALSE if this is not an OUTPUT statement. If, for some reason, the FRSTSPEC trigger is set (meaning we have not yet begun a group), we exit TRUE, without doing anything.

We put a closing right parenthesis on the ALPHALST. Then we file an ENDIOLNO (see Section 3.60.1.1). Now we want to put out the FORMAT statement that we have been accumulating on the ALPHALST. So we file a LABELNO followed by the label that was created by OPENOUT. Then we file a FORMATNO and call PKALPHML to pack up the ALPHALST, four characters per word, and file it on the CODELIST, preceded by a word count.

Finally, we fall into the entry point RESETOUT, which is also called initially and after unsuccessful nested I/O list TRYs. This zeros the ALPHACNT (there is nothing on the ALPHALST) and the PTRIG (no scale factor has been put out yet if a new FORMAT is begun), and sets FRSTSPEC TRUE (the next call on OPENOUT will be a first specification). Then we exit TRUE to signal that this was an OUTPUT statement.

3.66.2.4 FORMAT Spec By Mode (FSBYMODE)

This is called by CLOSEOUT. It puts onto the ALPHALST a FORMAT specification selected on the basis of the mode of the item pointer in WO. It does this by a M BAM on the FSTABLE (FORMAT Spec Table), which really does all the work. The "work" is to EAT an appropriate table entry that contains a FORMAT spec preceded by a character count. These specs are currently as shown in the first column below, but they should be changed to that shown in the second column, because the interpretation of the G format has been changed in the run-time just for this purpose (and it will shorten the generated output of OUTPUT statements as well as pulling code out of the compiler).

	now	could become
Integer	G	G
Real	G.6	G
Double	G.15	G
Complex	H(G.6-XH,G.6-XH)	H(G-XH,G-XH)
KOMPLEX	H(G.15-XH,G.15-XH)	H(G-XH,G-XH)
Logical	G	G

The table entry EATen will be used by LODALPHA (below) to put the FORMAT specification onto the ALPHALST. First, however, if this is a floating point item, we have to check whether a scale factor has been put out yet in this FORMAT (indicated by PTRIG). If not, we put out 1P and turn PTRIG on.

The table entry EATen will be used by LODALPHA (below) to put the FORMAT specification onto the ALPHALST. First, however, if this is a floating point item, we have to check whether a scale factor has been put out yet in this FORMAT (indicated by PTRIG). If not, we put out 1P and turn PTRIG on.

3.65.2.5 Load Alpha LODALPHA

Expects in the EATEMP the starting address of an entry of this form:

Word 1 : Character count

Words 2-n : The FORMAT specification characters (packed four per word)

It uses the machine language subroutine UNPACKML to get at the characters one at a time. It then picks up the Character Translate Table version of each character (see Section 2.4) and moves it onto the ALPHALST.

3.66 PAUSE Statement

3.66.1 PAUSE Statement PAUSESTA _ Pass 1)

Call Integer or Zero (INTORZ, below) to get an Integer constant or an alphanumeric string constant. File PAUSENO and a pointer to the constant. Exit through Active Exit.

Transmits two items to Pass 2:

1. PAUSENO
2. Constant to be displayed (pointer to Integer Constant List or String Constant List)

3.66.2 Integer or Zero INTORZ)

This routine, used by PAUSE and STOP, was originally written to scan for an Integer constant and, if none was present, to create the value zero. For IBM compatibility, it was later modified to also accept an alphanumeric string constant (surrounded by quotes). Note that Registered Signed Integer Constant Scan and Get Constant A (RSINTSCN and GETCONA, under Constant Scan) both scan and register a constant, but do not do the type check that is normally done in expressions.

3.66.3 PAUSE Statement Gen PAUSESTG _ Pass 2

There are two varieties of PAUSE statement, as seen both from the source language viewpoint and the object code. The more traditional form has an integer after the word PAUSE; the alternate form has a string constant--a message enclosed in quotes.

If an integer appears, Pass 1 registers it as an integer constant, and Pass 2 detects this by SOTing for ICONSCOD. If it is an integer constant, it generates a load of the value into register 9, followed by a

```
BAL,6          9PAUSE
```

The load into register 9 is usually immediate (i.e., if the number is less than 524288).

If a message appears rather than an integer, the location of the string constant is loaded into register 7, and then a

```
BAL,6          9PAUSEM
```

is generated.

3.67 PRINT Statement PRINTSTA _ Pass 1 only

File the PRINTNO (same as WRBCDNO), pick up in WO a pointer to the printer output unit number (108), and go to FORTRAN II Output Scan.

3.68 PUNCH TAPE Statement PCHTAPST _ Pass 1 only)

See the ACCEPT TAPE Statement, section 3.21, for a description of the ambiguity that also exists between PUNCH TAPE and PUNCH.

If PUNCH TAPE proceeds, file the PCHTAPNO (same as WRBCDNO), pick up in WO a pointer to the paper tape output unit number (104), and go to FORTRAN II Output Scan.

3.69 PUNCH Statement PUNCHSTA _ Pass 1 only)

Control comes here only if PUNCH TAPE fails (see explanation under ACCEPT TAPE and ACCEPT statements, 3.21 and 3.22). Rescan the word "PUNCH", file the PUNCHNO (same as WRBCDNO), pick up in WO a pointer to the card punch output unit number (106), and go to FORTRAN II Output Scan.

3.70 READ Statements

3.70.1 READ Statements READSTAS _ Pass 1)

If the first character after "READ" is not a left parenthesis, go to FORTRAN II READ Statements. If it is a left parenthesis, signal that this is a read statement (as opposed to a write statement), and fall into READ-WRITE Scan (below), which processes all BCD and binary READ and WRITE statements.

3.70.1.1 READ-WRITE Scan RDWRSCAN _

This is used by all the FORTRAN IV-type READ and WRITE statements. It scans for unit number, FORMAT reference, END=, and ERR=, and determines what indicator number to file according to which of those it finds, as shown below:

RD2CDNO	READ(unit,format)
BCDEENO	READ(unit,format,END= and/or ERR=)
RDBINNO	READ(unit)
BINEENO	READ(unit, END= and/or ERR=)
WRBCDNO	WRITE(unit,format)
WRBINNO	WRITE(unit)

There are one to four words of information following the indicator number, depending on which of those statements it is (see below). A unit number can be any numeric expression; if it is an Integer constant, it is checked for being in the range 0-9999. A FORMAT reference can be a statement number, an ASSIGNed variable, or an array name. This is handled by FORMAT Reference Scan (FORMREF). A routine called OSAEE is used to look for the END= and ERR=. This is more complicated than it ought to be, because they are allowed to appear in either order, but Pass 2 wants them sent with END= first. Each is followed by an active statement number, handled by Active Statement Number Scan. If one is present, but the other is missing, the absolute value zero is used instead of a label pointer.

If anything follows the right parenthesis (i.e. an I/O list), go to I/O List Scan. Otherwise, go to End of I/O List.

The number of items sent to Pass 2 is shown below for the various kinds of statements handled by this section:

Binary READ/WRITE 1. RDBINNO or WRBINNO

	2.	Unit number (pointer to expression)
BCD READ/WRITE	1.	RDBCDNO or WRBCDNO
	2.	Unit number
	3.	FORMAT reference (label, subscripted array plex, or symbol table pointer)
Binary READ with	1.	BINEENO
END= and/or ERR=	2.	Unit number
	3.	END= label (pointer to label list) or zero
	4.	ERR= label or zero
BCD READ with	1.	BCDEENO
END= and/or ERR=	2.	Unit number
	3.	FORMAT reference
	4.	END= label or zero
	5.	ERR= label or zero

3.70.1.2 FORTRAN II READ Statements F2RDSTAS _ Pass 1 only __

These statements illustrate well the syntax ambiguities that the TRY-Fail mechanism is designed to handle. Not only is

```
READ TAPE, X
```

a READ statement rather than a READ TAPE statement (TAPE is either an ASSIGNed variable or an array name), but

```
READ TAPE = 0
```

is neither a READ statement nor a READ TAPE statement. This is one of the few cases where there is a double ambiguity in determining what kind of statement it is. So, after "READ" is found, we set the Fail Exit to go to the READ Cards Statement if the other FORTRAN II READ types fail. If that happens, we will then set the Fail Exit back to Assignment Statement, in order to try that if READ Cards fails. The following four statements each look for a special quote after the word "READ".

3.70.1.2.1 READ INPUT TAPE Statement Pass 1 only)

File the RDINTPNO (same as RDBCDNO). Then call BCD Tape Scan, which files a unit number and a FORMAT reference. If there is an I/O list following, go to I/O List Scan; otherwise go to End of I/O List.

See FORTRAN II I/O Scan for a description of the items that are transmitted to Pass 2.

3.70.1.2.2 READ TAPE Statement Pass 1 only)

File the RDTAPNO (same as RDBINNO). Then call Binary Tape Scan, which files a unit number. If there is an I/O list following, go to I/O List Scan; otherwise go to End of I/O List.

See FORTRAN II I/O Scan for a description of the items that are transmitted to Pass 2.

3.70.1.2.3 READ DRUM Statement Pass 1)

File RDDRMNO. Then call Drum Scan, which files a unit number and a drum address. There must be an I/O list on a READ DRUM statement, so go directly to I/O List Scan.

Three items are transmitted to Pass 2 (not including the I/O list items):

1. RDDRMNO
2. Drum number (pointer to expression)
3. Drum address (pointer to expression)

3.70.1.2.4 READ DISC Statement Pass 1)

Allow alternate spellings "DISC" or "DISK". ("DISC" is preferred by Xerox.) File RDDSCNO. hen call Disc Scan, which files a unit number and a disc address. There must be an I/O li st on a READ DISC statement, so go directly to I/O List Scan.

Three items are transmitted to Pass 2 (not including the I/O list items):

1. RODSCNO
2. Disc number (pointer to expression)
3. Disc address (pointer to expression)

3.70.1.2.5 READ Cards Statement Pass 1 only)

Control comes here only after one of the other FORTRAN II READ statements fails (e.g. READ INPUT TAPE). Usually this happens immediately after the word "READ" when the following few letters do not satisfy any of the required quotes. We put the Fail Exit back to go try Assignment Statement in case this fails. Then we have to rescan the word "READ", since the fail and restore put us back at the beginning of the statement.

File the RDCRDSNO (same as RDBCDNO). THEN PICK UP IN WO a pointer to the card reader input unit number (105), and call FORTRAN II I/O Scan to process a FORMAT reference and determine whether there is an I/O list following. If there is, go to I/O List Scan, otherwise go to End of I/O List.

See FORTRAN II I/O Scan for a description of the items that are transmitted to Pass 2.

3.70.1.2.6 FORTRAN II I/O Scan F2IOSCAN)

Alternate entries: BCD Tape Scan (used by READ INPUT TAPE and WRITE OUTPUT TAPE)

Binary Tape Scan (used by READ TAPE and WRITE TAPE)

This routine is used by all the FORTRAN II input/output statements (both BCD and binary) except READ/WRITE DRUM/DISC. If it is entered through BCD Tape Scan or Binary Tape Scan, it scans for a tape unit number first, and, if it finds an Integer constant, checks it for being in the range 0-9999. If not called by a TAPE statement, there is already a pointer in WO to a unit number constant that was created by the statement. (E.g. PRINT creates 108.) If called by a BCD statement (as opposed to a Binary statement), it scans for a FORMAT reference (see FORMREF), which may be a statement number, an ASSIGNed variable, or an array name.

The particular statement has already filed an indicator number. This routine files the tape unit number and/or the FORMAT reference, tests whether there is anything further in the statement (i.e. an I/O list) and returns with the answer.

The items transmitted to Pass 2 are shown below for the various statements (this does not include I/O list items):

ACCEPT, ACCEPT TAPE,	1.	RDBCDNO
READ, READ INPUT TAPE	2.	Unit number (pointer to expression)
	3.	FORMAT reference (pointer to label table, symbol table, or subscripted array plex)
TYPE, PUNCH TAPE, PUNCH,	1.	WRBCDNO
PRINT, WRITE OUTPUT TAPE	2.	Unit number
	3.	FORMAT reference
READ TAPE	1.	RDBINNO
	2.	Unit number
WRITE TAPE	1.	WRBINNO
	2.	Unit number

3.70.1.2.7 Drum Scan and Disc Scan (DRUMSCAN and DISCSCAN)

Although large portions of these routines are written separately, they do exactly the same thing and ought to be combined into one to save code. I suspect they were separated originally on the possibility that the error check for drum-address-in-range and disc-address-in-range would be different tests, but in fact they are the same.

Each routine scans for two expressions, the first representing the drum/disc number, and the second the drum/disc address. If these are Integer constants, they are checked for being in range. The two expression pointers are filed on the Code List. Then it makes sure there is an I/O list following the address expression (since a disc or drum operation without an I/O list is meaningless). If the list is missing, it gives an error message and fails.

3.70.1.3 FORMAT Reference Scan FORMREF

Used by all I/O statements that reference a FORMAT. A FORMAT reference can be:

1. A statement number (local or non-local)
2. An array name (the array contains the FORMAT), or
3. A variable (which has been ASSIGNed a FORMAT number)

If the first character is not a letter, we assume it is a digit and call Statement Number Scan. Then we set up the label trait REFFORM (referenced as a FORMAT), so that if the label shows up later on a statement other than a FORMAT, we can give a diagnostic (see Non-FORMAT Label, 2.9.9.8). Conversely, also check to see if the label has already appeared on a non-FORMAT statement, as indicated by the DEFEXEC trait. If it has, give a warning message.

If the first character was a letter, we have to determine whether this is an array name standing alone or a variable (which might be a scalar name standing alone or it might be a subscripted array element). Unfortunately, this has to be done with a TRY, because once we have scanned an array name and then determined that there is a left parenthesis after it, it is too late to get down into the middle of Any Mode Variable Scan at the point where it has just found the name of an array element. So Array Name Scan is TRYed, and if it finds an array name followed by a comma, right parenthesis, or end-of-line, it succeeds; otherwise it fails, the scan pointer is returned, the Symbol Table may have to be restored, and we start all over by calling Any Mode Variable Scan. It would seem as though Array Name Scan could safely succeed if it found a scalar standing alone too (which it does not); all that would be necessary would be to check for the permissibility of SCALAR and then also do a SUP and REG of that.

3.70.1.4 Array Name Scan -ARYNAMSC _

Used in I/O lists, in ENCODE and DECODE (for the buffer location), and in statements that require a FORMAT reference (which can be an array name standing alone). It must be called with a TRY, because once it has scanned a name and found either that it does not stand alone or that it is not an array, it is too late to pick up somewhere else with the name we have already scanned. This is one of the drawbacks to a top-down expression scan (see Section 2.7.3).

A TRYed routine must save all the lists it thinks it might change. The only one in this case is the Symbol Table. If it scans a name successfully, it checks for comma, right parenthesis, or end-of-line, which would establish that the name is standing alone. If this fails, or if the name is not an array, the routine fais. Otherwise, go to Register Type to establish a type for the array.

3.70.2 BCD READ Statement Gen BCDREADG _ Pass 2)

This is an easy statement to generate. We have a unit number and a format to handle and a library routine to call. With the unit number we do a little cleverness, and in case it is 105 we save one word of generated code by calling 9READ instead of 9BCDREAD. (9BCDREAD expects the unit number in register 9, whereas 9READ knows that unit 105 is intended.)

We JRS to NXINWORD for the unit number, then QJS to CHEKUNIT which, among other things (see 2.23), checks to see if it is 105; if it is not 105, we guarantee it Integer in AC (register 9). We then call FORMGEN (Format Gen) which calls NXINWORD for the FORMAT pointer, then generates appropriate code to get its address into register 7. Then we generate a BAL,6 to either 9BCDREAD or 9READ. Then we Reserve the Delayed Code List (DCODLIST)--see 3.60.2.3 to find out why--and exit through Statement Exit (STATEXIT).

3.70.3 Binary READ Statement Gen BINREADG _ Pass 2)

The Binary READ statement is generated almost exactly the same way as REWIND, ENDFILE, BACKSPACE, and SENSE LIGHT. This similarity occurs because, to Pass 2, the Binary READ statement is just the front part of a source language Binary READ statement: i.e., the items in the I/O list are seen by Pass 2 as if they were separate statemments.

Therefore, the only job is to evaluate an Integer expression for the unit number (leaving the result in register 9) and gen a call on 9BINREAD (a library routine). There is one small difference between Binary READ/WRITE and the other statements mentioned above: they reserve the Delayed Code List (DCODLIST) before doing their work. All the I/O statements involving an I/O list do this. The reason is that the ENCODE and DECODE statements sometimes generate some code that needs to be put out at the end of the I/O list (just after the call to 9ENDIOL). No other statements need to do this, and even ENCODE and DECODE do it rather rarely; but when Pass 2 receives an End I/O List Indicator from Pass 1, it does not know (or care) what kind of statement is having its list ended. Therefore, it always takes the bottom file from the DCODLIST and outputs it. The file is, of course, usually empty.

When I say it "outputs" the file of code, I really mean that it copies it onto the bottom of the CODELIST (without its Reserve mark). Then the normal mechanism of Assemble Code (ASMCODE) puts it on the OUTLIST.

To sum up, Binary READ Statement Gen (BINREADG) ADRs a 9BINREAD link, then joins code shared by Binary

WRITE Statement Gen (BINWRITG) which reserves the CDODLIST and then joins code shared by REWIND, ENDFILE, BACKSPACE and SENSE LIGHT, which gets an Integer expression pointer off the INPUTLST, gets it evaluated with GTACUINT, then generates a library link to P0, then discards P0 and exits through STATEXIT.

3.70.4 BCD READ with END=/ERR= Statement Gen BCDRDEEG _ Pass 2)

BCDRDEEG is much like BCDREADG. It is simpler in that no test is made to see if the unit number is 105 (actually, the test for unit 105 is made, but the results of the test are ignored). Then there are two extra words (PZE's) generated in the calling sequence, to handle the two labels. And of course we call a different library routine-- 9BCDRDEE instead of 9BCDREAD or 9READ.

The code for generating the labels is in a routine called ENDERRGN (END/ERR Gen) which is also shared by the Binary READ with END=/ERR= Statement. We simply JRS to NXINWORD for each label and, if the pointer is nonzero, we generate a PZE to the label. If the pointer is zero, indicating the absence of the label in the source statement, then we generate a PZE 0. The comments "LABEL FOR END=" and "LABEL FOR ERR=" are appended to the generated lines when actual labels appear.

3.70.5 Binary READ with END=/ERR=Statement Gen BINRDEEG _ Pass 2)

BINRDEEG is much like BINREADG. We generate a call on 9BINRDEE with the unit number in register 9. Following the BAL,6 are two PZEs for the two statement labels involved with END= and ERR=. The generation of these labels is done in ENDERRGN, described in 3.70.4.

3.70.6 READ DRUM Statement Gen RDDRUMSG _ Pass 2)

READ DRUM is just like, say, Binary READ, except that in effect it has two unit numbers instead of one. These are called the Unit Number and the Address and are sent to the library routine in registers 9 and 11, respectively.

We ADR the 9RDRUM Link and branch to READ/WRITE DRUM/DISC (RWDRMDSC, below) which does all the work.

3.70.7 READ/WRITE DRUM/DISC RWDRMDSC _ Pass 2)

This routine is hsared by four statement generators: READ DRUM, WRITE DRUM, READ DISC (or DISK), and WRITE DISC (or DISK). It gets two expression pointers via NXINWORD, guaranteees the second one Integer Addressable (for later loading into register 11), then guarantees the first one Integer in Accumulator (register 9). Then it builds an appropriate Load Word or Load Immediate instruction for the second expression, builds a BAL,6 thru P0, gets rid of P0, Reserves the Delayed Code List (3.60.2.3 tells why) and goes to Statement Exit (STAEXIT).

The process for getting the second operand loadeed is handled with a not-wholly-appropriate subroutine called LOAD2XI (Load 2nd Operand as for X**I) that is also used for raising things to powers. It is clumsy, but it works.

3.70.8 READ DISC Statement Gen RDDISKSG _ Pass 2)

READ DISC is just like, say, Binary READ, except tha in effect it has two unit numbers instead of one. These are called the Unit Number and the Address and are sent to the library routine in registers 9 and 11, respectively.

We ADR the 9RDDISC Link and branch to READ/WRITE DRUM/DISC (RWDRMDSC, 3.70.7) which does all the work.

3.71 REAL Statement REALSTA _ Pass 1 only)

See Type Statements, Section 3.80.

3.72 REPEAT Statements

3.72.1 REPEAT Statements REPEATST _ Pass 1)

Like the DO statement, the REPEAT statement creates a three-item entry on the DO list, for later use by Process Ends of DOs. The first two items are the same as for DO, namely the terminal statment number (scanned by DO Statement Number Scan) and the current line number (derived by Derive Line Count ML). The first enables us to know when we have reached the statement that ends the REPEAT range, and the second enables us to output better diagnostics if the REPEAT is never closed (see Check For Unclosed DO-loops, under Finish Up, 2.9.4). The third item, for DO loops, is the control variable. This is used in handling subscripts inside DO loops. It is important to know whether a subscript inside a DO loop is dependent only on control variables of current DO loops, since if it is, it can be remembered across statement boundaries, since control variables are not allowed to be modified within a loop. However, this is not true of REPEAT loops. These are completely dynamic; the control variable can be changed, if desired. Therefore, when looking for control variable subscripts, we specifically do nto want ot know about REPEAT loop control variables. We still need the first two items on the DO List, in order to close the REPEAT loop, but the third item is set to zero, so that it will never be recognized as a control variable.

A comma is allowed to follow the statement number for REPEAT WHILE. A comma must follow the statement number for REPEAT FOR. So, if there is no comma, we go try REPEAT WHILE.

3.72.1.1 REPEAT WHILE Statement Pass 1 _

File a REPWHINO and a Logical expression pointer. Note that we call Logical Expression at a special entry point that does not save numerous list data. The other entry point, LOGEXP, is always TRYed (when you are not sure if it is going to be a logical expression or not). Exit through the DO Exit.

Two items are transmitted to Pass 2:

1. REPWHINO
2. Pointer to Logical expression

In addition, three items are placed on the DO List:

1. Terminal statement number
2. Current line number
3. Fake control variable (absolute value zero)

3.72.1.2 REPEAT FOR Statement Pass 1

File REPFORNO. Call Any Mode Variable Scan. (Note that a REPEAT FOR control variable does not have to be just Integer, Real, or Double Precision; it can be Complex or Logical, but if it is, it cannot have any "triples"; see below.) Call Outer DO Check, to see if there are any currently open DO loops using this same control variable. (REPEATs can check for outer DOs, but DOs cannot check for outer REPEATS.) Now call Set Mode, which puts in PO an indicator of what types (modes) of expressions will be allowed on the right of the equal sign (same as in assignment statements).

Now we are going to scan the expressions and/or triples appearing to the right of the equal sign. We will accumulate them on the ARGLIST and then build a REPEAT List Plex from them. If the control variable is Complex, Komplex, or Logical, it cannot have triples only expressions. The expressions are handled by Expression Scan, which knows from the indicator in PO what types it is allowed to find.

Triples are considerably more complicated, because of syntactic ambiguities. For example, (X,Y) is clearly a triple (with the third expression missing). On the other hand, (3,4) could be a triple or a complex constant; we obviously would rather assume it was a triple. But then, (3,4)*CX can only be a complex expression, not a triple. Fortunately, all of this does not really apply, since only Integer, Real, and Double Precision control variables can have triples, and they cannot have complex expressions. However, there is still the lesser ambiguity that (X,Y) is a triple, while (X)*Y is not, and our top-down approach is not able to cope with this on the fly.

So, if an item begins with a left parenthesis, we TRY Triple. It calls REPEAT Control, which is a sub-entry to DO Control, which scans for the two or three expressions and builds a triple plex from them. If there is then a right parenthesis, as expected, Triple has succeeded. If not, it has failed, but since we have already found at least two expressions, separated by commas, there is nothing else it can be either. So we set the Automatic Succeed Flag

(on the bottom of the TRY List) to assure that the TRY really fails; that is, it does not just come back FALSE so that something else is tried; it comes back and fails at the upper level. (See TRY-Fail, 2.7.)

If the TRY comes back FALSE, or if the item did not begin with a left parenthesis, we just scan an expression, as above. Note that REPEAT Control puts its triple plex on the Code List. We would rather have it on the ARGLIST, so we have to move it over. In order to combine this code with that for expressions, we wastefully move the expression pointer over to the Code List, and then fall into the code that moves it right off again and onto the ARGLIST.

When all the items have been scanned, we build a REPEAT List Plex from them and file it on the Code List. Then we exit through DO Exit.

Three items are transmitted to Pass 2:

1. REPFORNO
2. Control variable (Symbol Table pointer)
3. REPEAT List Plex pointer.

In addition, three items are placed on the DO List:

1. Terminal statement number (pointer to label list)
2. Current line number (value, not pointer)
3. Fake control variable (absolute value zero)

3.72.2 REPEAT WHILE Statement Gen REPWHSTG _ Pass 2

Code for the REPEAT WHILE statement is generated in two parts: (1) the loop top code, placed in the CODELIST contains the Logical expression evaluation and conditional branch to the bottom (outside) of the loop; (2) the loop end code, placed on the Delayed-Code-List (DCODLIST) for later inclusion in the code stream, merely contains a branch to the top of the loop, and the definition of the loop bottom label. For example, the statement

```
REPEAT 23, WHILE I .LT. 200
```

produces in the code list

```
1G   LW,9 I
```

```
        CI,9 200
        BGE     2G
```

and, on the Delayed Code List,

```
        B       1G
2G      EQU     S
```

The code generating routine (REPWHSTG) first JRSs to SCRPTGEN to make sure that any pending subscript calculation code is put where it belongs, and that subscript code required for evaluation of the Logical expression will be placed inside the loop (c.f., section 3.3). Then REPWHSTG calls MAKELBL to obtain the label pointer corresponding to 1G in the above example, and a copy of it is placed in LABEL1. Next the logical expression pointer is obtained from the INPUTLST, the logical exprssion is simplified (via SIMPEXP), a label pointer (corresponding to 2G above) is obtained and placed on the Special Label List (SPLBLIST) by MSPECLBL, and EXPJUMPF is called to generate expression evaluation and conditional branch code. This branch says, in effect, "If the Logical expression is false, branch to 2G". The loop top code is now complete; we RSV the CODELIST to separate the two pieces of code, BIN a branch to G (the pointer to which is still in W0), gen a label definition for 2G (removing it from the SPLBLIST), and copy the current file of code from CODELIST to DCODLIST. We now go back and define the top of loop label 1G. This is done in a somewhat unusual manner, in order to insure that the loop includes any "retroactive subscript evaluation code" (see 2.14.4) that may have resulted from generating the Logical expression. First, we call SCRPTGEN, which places such script code up ahead of the loop top code on the CODELIST; then we copy the entire file of code to DCODLIST gen the definition of 1G (on CODELIST, of course), then copy that file back from DCODLIST to CODELIST.

Finally, we RSV the DOLIST thus supplying a dummy file of DO-register information which will be required by End of Loop Gen (ENDLOOPG; see 3.38.3).

3.72.3 REPEAT FOR Statement Gen REPEATSG _ Pass 2)

Consider the statement

```
REPEAT 23, FOR V = Q  , Q  , ..., Q
```

where each Q; is either a single expression, or a triple of the form (A, B, C). If n=1, the REPEAT loop will be generated as

```
        START Q
        ...                     (statements in the REPEAT
```

range)

```
                    END Q
```

If n 1, the statements in the REPEAT range are placed in a subroutine called by the loop-controlling code as follows:

```
                    START Q
                    BAL,15      1G
                    END Q
                    START Q
                    BAL,15      1G
                    END Q
                    . . .
                    START Q
                    BAL,15      1G
                    END Q
                    B           2G
1G                  STW,15      1PTMP
                    . . .                   (statements in REPEAT range)
                    B           *1PTMP
2G                  EQU         $
```

In the above, START Q; and END Q; represent loop initialization and control code for the item Q; as follows:

(1) If Q; is a single expression, say A, then there is no END code, and the START code is equivalent to the code generated by the assignment statement V=A.

(2) If Q; is a triple (A, B, C) where C is not a constant, the START and END code behaves as if it were written:

```
                    V = A                   (START code)
                    CTEMP = C
```

```
                GO TO 2
1               . . .
                CTEMP= C                    (END code)
                V = V+CTEMP
2               BTEMP = B
                IF (CTEMP .GE. 0 .AND. V .LE. BTEMP  .OR.
               X    CTEMP .LT. 0 .AND. V .GE. BTEMP) GO TO 1
```

In this code, the step value C is recomputed each time through the loop and placed in a temp (CTEMP); B is also reevaluated, and left in a register (as BTEMP above) during the tests indicated in the last statement.

(3) If Q is a triple (A, B, C) with C constant, the generated code is simpler. This case is broken down four sub-cases according to the sign of C, and according to the "conditionality" of loop execution; in this regard we call a loop "unconditional" if, at compile-time, we can determine that it will be executed at least once.

(3a) Increasing, conditional case (C 0; either A or B non-constant, or both constant with A B): code is generated as follows:

```
                V = A
                GO TO 2
1               . . .
                V = V+C
2               IF (V .LE. B) GO TO 1
```

(3b) Decreasing conditional case (C 0; either A or B non-constant, or both constant with A B): same as abvoe, except the last line is

```
2               IF (V .GE. B) GO TO 1
```

(3c) Increasing unconditional case (C 0; A and B constant with A B): like (3a), except that there is no GO TO 2:

```
                V + A
1               . . .
                V = V+C
```

IF (V .LE. B) GO TO 1

(3d) Decreasing unconditional case (C 0; A and B constant with A B): like (3b) without the GO TO 2, as above.

Please refer to the comment preceding REPEATSG in the PASS2F listing for examples of actual machine code generated in instances of these cases.

At the upper level, REPEATSG is concerned with the overall order of START/END code generation. It brings in the Repeat List Plex Pointer and PULs its items (pointers for Q , ..., Q) onto the Repeat List (REPLIST); if there is only one item, it is handled by Single Repeat Item Gen (SREPGEN); this is the same routine used by IODOGEN to handle implied DO-loops in I/O lists. SREPGEN calls Repeat Item Gen (REPITEMG) to generate one file of code on the Code List (the START code) and one on the Delayed Code List (for the END code). In case there is more than one repeat item, REPEATSG creates a label to be used for the subroutine containing the statements in the repeat range; then, for each repeat item, it calls REPITEMG to gen START and END code files and puts them together (on the Code List) with a BAL to the subroutine between them. When all the repeat items have been processed in this way, the Code List contains all the loop control code, and nothing has been left on the Delayed Code List. Finally, we generate the branch around the subroutine, and we gen a definition of the subroutine label followed by STW,15 into the link temp; we also generate a small file of code on the Delayed Code List for inclusion at the end of the REPEAT range -- it contains the subroutine exit instruction (B *linktemp) and the definition of the branch-around label. An empty file is placed on the DO List (DOLIST) so that ENDLOOPG will have something to remove (c.f., 3.38.3).

Repeat Item Gen (REPITEMG) is the subroutine which generates the START and END dode for one of the repeat items, Q; , according to the case breakdown given above. It generates the START code (preceded by its "retroactive" subscript generation code, if any) on the Code List, and the END code (with its script code) on the Delayed Code List; nothing is left on the Script List. An empty file is placed on the DO-List for reasons explained in 3.38.3. If the repeat item is a simple expression, REPITEMG simply calls IASSGEN to generate the assignment code for V=A, then calls SCRPTGEN to combine any subscript code that may have been produced; it then RSVs the Delayed Code List and DO List. If the repeat item is a triple (represented by a Repeat Triple Plex pointer), REPITEMG PULs the plex to obtain its constituents, A, B, and C, generates the initial assignment code (V=A), then determines whether C is a constant. We arrive at the label GOODTRIP in case C is constant (corresponding to case 3 above), or BADTRIP in case C is not constant (case 2). In the latter case, we allocate an Integer temp to hold the step sign (CTEMP); an integer temp is sufficient even if V is double precision, since all we care about is C's sign. We also gen the code to evaluate C and store its sign in CTEMP. For a <u>good</u> triple, on the other hand, we call NOTIMEST to determine whether the triple is conditional. For a messy (non-constant C) or conditional triple, we create a label (label 2 in the examples) and generate a branch to it. We now define the top-of-loop label (label 1 above), and call SCRPTGEN to cause retroactive script code to be placed ahead of the START code. This concludes generation

of the START code; we RSV the Code List to prepare it to accumulate the END code in two parts: "step" code, and "test" code. The step code for a messy triple behaves like the statement

```
V = V+(CTEMP=C)
```

except that the store to CTEMP is only a one-word store, regardless of C's type. For a good triple, the step code is simpler:

```
V = V+C
```

After generating one of these code sequences, we define the test-point label (label 2 above), if it exists, and call SCRPTGEN to associate script code with the step code. Test code generation begins with a RSV of the Code List, so that any retroactive script code required by the test code will be placed at the beginning thereof, rather than ahead of the step code. The test code begins with code to evaluate B in the accumulator. For a messy triple it continues as follows:

```
        LW,15     CTEMP
        BGEZ      3G
        CW,9      V          (or CW,8 or CD,8)
        BLE       1G
        B         4G
3G      CW,9      V          (or CW,8 or DC,8)
        BGE       1G
4G      EQU       $
```

The test code for a non-messy triple is:

```
        CW,9      V          (or CW,8 or CD,8)
        BGE       1G         (for C 0; BLE for C 0)
```

In either case, we finish up by calling SCRPTGEN to associate any retroactive script code, moving the END code file to the Delayed Code List, and RSVing the DO List.

3.73 RETURN Statement

3.73.1 RETURN Statement RETURNST _ Pass 1)

The RETURN statement is complicated immensely by the necessity of acknowledging IBM's form of non-standard return. First of all, we have to accept "RETURN 3" , where 3 is the asterisk count. Furthermore, the statement "RETURN J" is ambiguous. J can be either an asterisk count or a perfectly normal (and recommended) Xerox Assigned Return, and we may not be able to find out until run-time.

It would be nice if a "RETURN 3", in a subprogram with no asterisks (or fewer than three), could be diagnosed at its appearance in Pass 1. Unfortunately, the asterisks may show up in a later ENTRY statement, so Pass 2 has to do the diagnosis. (See ENTRY Statement, 3.47, for further discussion of this.) Therefore, we send Pass 2 the current line number for each non-standard RETURN.

If the RETURN appears in a main program, we give a warning, and Pass 2 generates a STOP instead of a RETURN. Exit through Jump Exit.

For an ordinary RETURN, one item is transmitted to Pass 2:

1. RETURNNO

For an assigned return, two items are transmitted to Pass 2:

1. ASRETNO
2. Pointer to Integer constant or any mode varaible

and Pass 2 has to decide whether to generate an Assigned GO TO or an asterisk non-standard RETURN. If a variable has appeared in the RETURN statement, and there have been asterisks in a dummy list somewhere in the program, then Pass 2 cannot decide either, and it has to pass the dilemma on to the run-time.

3.73.2 RETURN Statement Gen RETURNSG _ Pass 2)

The flavor of code generated for RETURN dependw on the sequence of subprogram opening statements thus far encountered, as well as upon the Debug (DB) and Real Time (RT) option specifications; there is no additional information from Pass 1 accompanying the "RETURN Statment" indicator. If SPROGTRG is reset (indicating no SUBROUTINE or FUNCTION statement has yet appeared), we generate a "STOP 0" and exit:

```
LI,9     0

BAL,6        9STOP
```

On the other hand, if we <u>are</u> in a subprogram, and if FUNCTRIG is set (a FUNCTION statement has appeared more recently than any SUBROUTINE statemnt),

code is generated to load the accumulator from the variable corresponding to the function name, a pointer to which had been saved (by FUNCSTAG) in SPNAMEVP (Sprog Name Variable Pointer). Next, if the Real Time Sprog Trigger (RTSPGTRG) is set, indicating that the RT option was specified, and that this is the first subprogram of a subprogram compilation, then we generate the Real Time subroutine exit sequence:

```
LW,15       linktemp
LI,4        1
STH,4       1RTMP
```

where "linktemp" is the address of the Private Temp in which the return link was stored. Next, if the Debug option (DB) was specified, the "RETURN Check-in" sequence is generated:

```
BAL,6           9DBFHRS
GEN,1,14,17     0,line,label
```

where "line" and "label" are the current line number and label pointer. Finally, we generate a branch indirect to "linktemp" or register 15 (in the Real Time Sprog case).

3.73.3 Asinine RETURN Statement Gen ASRETSTG Pass 2

This routine is invoked during Pass 2 in response to a statement like

```
RETURN 5
```

or `RETURN V`

The first example makes sense only when within a subprogram whose header contains at least five asterisks in the dummy list; it means "return to the label argument corresponding to the fifth asterisk dummy." The second example has two possible meanings, depending upon whether there were any asteriks dummies in the subprogram header. If there were, then the example statement is interpreted, "return to the label argument corresponding to the Vth asterisk dummy"; otherwise, it is interpreted as "return to the dummy V", like GO TO V.

Additional information is supplied with the "Asinine RETURN" statement indicator: a variable or constant pointer representing the RETURN parameter, and the line number of the RETURN statement for error reporting purposes. An additional value, the number of asterisk dummies in the subprogram header, is available in the cell ASTCOUNT, having been established there by the SUBROUTINE or FUNCTION Statment Gen routine. If SPROGTRG is reset (i.e., no subprogram header has appeared), we branch to RETURNSG to gen an ordinary RETURN (which is STOP 0 in this case). An ordinary RETURN is also generated

for RETURN 0. If SPROGTRG is set, and the parameter is a nonzero constant, N, then N 0. If N ASTCOUNT, we generate code to fetch (to register 7) the appropriate word from the asterisk return vector, a list of length ASTCOUNT immediately following the return link temp in temp storage. These cells contain, at run-time, the argument words corresponding to the asterisk dummies. Next we generate special code that may be required for Real-Time and Debug applications, as described above for ordinary RETURNs except that 9DBFHRJ is used instead of 9DBFHRS; finally, a BAL to 9ASGOTO is generated. If N ASTCOUNT, we print an error message:

RETURN ON LINE xxx HAS ASTERISK COUNT

THAT IS TOO LARGE. STANDARD RETURN USED.

where "xxx" is the line number of the RETURN statement provided by Pass 1. After printing the message, we branch to RETURNSG.

If the RETURN parameter was a variable, and if ASTCOUNT=0, we treat the statement like gGO TO V by generating LI,7 V followed by Real-Time, Debug, and Assigned GO TO code, as above. If ASTCOUNT 0, the RETURN is handled by a special run-time subroutine, 9ASININE (c.f., 3.73.4); we generate:

```
LI,7      V
LI,5      linktemp  LOCATION OF RETURN VECTOR
LI,9      astcount  MAXIMUM NUMBER OF ASTERISKS
(Real-Time Sprog code, if any)
(Debug check-in via 9DBFHRJ, if required)
BAL,6          9ASININE
```

3.73.4 RETURN Statement at Run-Time 9ASININE)

When we encounter a statement such as

RETURN N

(i.e., a variable, not a constant), and there have been asterisks in a dummy list, we do not know whether the N is an ordinary ASSIGNed variable (in which case the statement has the same effect as GO TO N) or an IBM asterisk count (such as RETURN 3, meaning return through the third asterisk). This would be completely ambiguous, were it not for the fact that it can be determined at run-time. An asterisk count will be a small integer value, whereas an ASSIGNed variable will contain a Branch instruction (op-code X'68').

Therefore the decision making is passed on to a library routine called 9ASININE (ASterisk INterpreter for IBM Non-standard Exits). We provide this routine with the location of the asterisk vector, so that it can return via there if necessary. But if the variable contains a branch instruction (as it should in any good clean American coding), then 9ASININE will act just like 9ASGOTO (the ASSIGNed GO TO library routine).

3.74 REWIND Statement

3.74.1 REWIND Statement REWINDST _ Pass 1)

There may be any number of units specified, separated by commas. Each unit may be represented by an expression whose mode can be converted to Integer. For each unit, file the REWINDNO followed by a pointer to the expression. Exit through Active Exit.

If more than one unit is specified, they will be transmitted to Pass 2 as separate REWIND statements. Each one contains two items:

1. REWINDNO
2. Unit number (pointer to expression)

3.74.2 REWIND Statement Gen REWINDSG _ Pass 2)

In addition to the statement indicator, Pass 1 sends an expression pointer for the unit number to be rewound. With this pointer in WO we simply JRS to Grntee Integer In ACcumulator (GTACUINT), then build a library link to 9REWIND.

In fact, REWIND Statement Gen shares code with several other easy statement generators, so it starts out by ADRing the 9REWIND Link, then later generates the link through PO.

3.75 SENSE LIGHT Statement

3.75.1 SENSE LIGHT Statement SNSLITST _ Pass 1)

There may be any number of sense lights specified, separated by commas. Each sense light may be represented by an expression whose mode can be converted to Integer. If the expression is just an Integer constant, check that it is within range (0 through 24), and if not, diagnose and discard it. Otherwise, file SNSLITNO and a pointer to the expression. Exit through Active Exit.

If more than one sense light is specified, they will be transmitted to Pass 2 as separate SENSE LIGHT statements. Each one contains two items:

1. SNSLITNO
2. Sense light number (pointer to expression)

3.75.2 SENSE LIGHT Statement Gen SLITESG _ Pass 2)

SENSE LIGHT is another easy statement to generate, like REWIND, ENDFILE, and BACKSPACE. This is true even through the other statements are Input/Output statements, which SENSE LIGHT is not. But the generation is the same: evaluate an Integer expression (or at least guarantee the result Integer) and, with the result in register 9, call a library routine to do the work.

In the compiler, we ADR the 9SNSLITE Link, then branch to common code that is shared by the above mentioned statements. See the writeup for BACKSPACE Statement Gen, Section 3.26.2.

3.76 Statement Function

3.76.1 Statement Function Definition ASFD _ Pass 1

There has been historically considerable variation as to what these are supposed to be called. Originally, IBM called them Arithmetic Statement Functions (functions defined in a single statement.) Digitek, when they wrote the SDS 900 Series FORTRAN II, decided that they were statements more importantly than they were functions, so called them Arithmetic Function Definition Statements (statements that define an arithmetic function.) The abbreviated form, AFDS, came into some popularity. We prefer to call them Statement Functions, so that they can be referred to in documentation. For example, "This call can be satisfied by a FUNCTION subprogram, an assembly language subprogram, or a statement function." The other terminology makes it difficult to distinguish between the statement that defines a function, and what kind of a function it is. In our terminology, the function is a

Statement Function and the statement is a Statement Function Definition. The word "arithmetic" has to go, because that precludes "logical" type, and it is perfectly legal to have a logical statement function. Nonetheless, our listings (which contain much material intact from 9300 FORTRAN IV) still often refer to them as Arithmetic Statement Function Definitions, abbreviated ASFDs.

Control comes here only if the "quote" statements and the assignment statement have failed. Since statement functions are not very frequent (compared to assignment statements), we have no particular inclination to recover from errors and succeed in scanning further along the statement. Since the error messages that finally get printed are those that pertain to the type of statement that got the furthest through the scan, we would just as soon have the messages reflect "assignment statement" as "statement function".

So, (since statement functions are supposed to come at the beginning of the program), we test whether there have been any executable statements yet. If there have, we will not even scan for statement function; we just fail immediately. Note, however, that internal subprograms reset the Executable Statement Trigger, so that statement functions may appear again, at least until another executable statement shows up.

Then we get the function name, check for the permissibility of Subprogram Definition, and fail if not permissible. Otherwise, we set up new permissibility and register the (SOF) trait SPROGDEF, and also the (SOT) trait SPROG (Subprogram). Please refer to the description of the FUNCTION Statement for an explanation of this. We call Register Type to establish a type for the function. (This will be implicit type, unless the function's name has previously appeared in a type statement.) And we call Set Mode to establish the types that will be permissible on the right side of the equal sign. (E.g., Complex=Integer is valid, but Integer=Complex is not.)

If the function name has previously been declared EXTERNAL, and we are in real-time (i.e. the RT option was specified), we give a warning, because statement functions, like internal subprograms, are not re-entrant and should not be called by the outside world. In any case, we file an ASFDNO and the pointer to the function name.

Now comes some more of the interesting fiddling that was discussed under FUNCTION Statement and Allow Dimensions. Whereas within a FUNCTION subprogram, the name of the function must be considered a scalar variable, within a statement function, the name of the statement function must be considered an illegal reference (since we cannot implement recursive calls). I do not remember if there is any reason we could not just REG the name as ILLEGAL, and then REG it the way we want when we are done (the way we do with array names in Allow Dimensions). What we do instead is to create a new name (using CREATEID), mess up the old one temporarily so it will not be recognized, and REG the new name as ILLEGAL. There is also a SUP of permissibility for ASFUNC, and I suspect that it has no effect (and is probably a hold-over from 9300 FORTRAN IV), since I do not believe there is any place that would look at that permissibility prior to finding out that the name is registered as ILLEGAL.

Now we are going to scan the dummies and pile them up on the Work List, so we can build a Dummy Group Plex from them. Note, relative to the check of permissibility to be a statement function dummy, that the only thing that disallows this is having already appeared as a dummy in the same statement function. Otherwise, statement function dummies are allowed to be the same name as any other name in the program: scalars, arrays, subprograms, COMMON blocks, dummies in other functions, etc. But you cannot write:

```
F(X,Y,X) = expression
```

and this is diagnosed. In order to allow these dummy names to be the same as other names, we have to create separate Symbol Table entries for them. (This is for several reasons. For one thing, we want them to be able to have their own traits and permissibility. For another, they each have a location, which is different from the location for the other entity with the same name, and these locations must be available to Pass 2 to generate code and to Pass 3 to list them with the proper locations, both in the object listing and in the summary.) So we use two subroutines, Create New ID Entry (CREATEID) and Dekludge, which are described below. CREATEID creates a new entry for the dummy name found. If that name had not appeared previously, it still creates a new entry. The old entry will then reflect UNUSED class. Then, if it does subsequently appear elsewhere in the program, it will be updated appropriately. If it never appears elsewhere, it will be listed in the summary as an extraneous UNUSED identifier. (This is kind of unfortunate, and we would be happier if it did not happen, but it is difficult to do anything about it.) CREATEID also temporarily destroys the old ID entry, so that Symbol Table searches will not find it.

After scanning the statment function, Dekludge will be used to restore the old ID entry so that both are correct. However, the old one appears first in the Symbol Table and will therefore take precedence. Consider also the following case:

```
DIMENSION X(10)

F(X) = something

G(X) = something else
```

First X, is an array. Then it becomes a statement function dummy, and CREATEID provides a new Symbol Table entry. Then it is used again as a new statement function dummy, which also has to have a unique Symbol Table entry. Therefore, CREATEID must be prepared to temporarily destroy any number of previous "old" entries for a name, and Dekludge must be prepared to put them all back.

Each dummy has the permissibility of ASFDUMMY set up and is registered as a SCALAR (class) and DUMMY (trait). And it is given implicit type immediately, if it has not already been declared explicitly in a type statement.

After all the dummies have been collected on the Work List, we build a Dummy Group Plex from them and file it on the Code List. Then we go scan the expression on the right side of the equal sign. Remember that Set Mode has established the types that will be allowed in this expression, and an indication of this is lurking on the bottom of the Parameter List (i.e. in P0). We file the expression and check for excess right parentheses. Make sure this is the end of the statement before bothering to restore the old ID entries. Exit through Declarative Exit 1 (invalid in BLOCK DATA -- statement functions are not executable in the normal sense and are thus much like declarative statements).

Four items are transmitted to Pass 2:

1. ASFDNO
2. Function name (pointer to Symbol Table)
3. Dummy group plex pointer
4. Defining expression (plex pointer)

3.76.1.1 Create New ID Entry CREATEID

This is used by Statement Function Definitions and by FUNCTION and SUBROUTINE statements, where there has to be more than one entity with the same name. Given a pointer (in W0) to an identifier, it creates a new Symbol Table entry for the same name, inheriting only the type of the old one (if it had one). Then it clobbers the old entry (at least to the point where it will never be recognized by a Symbol Table search), and any other old entries that may have previously been created (see Statement Function Definition), by storing all ones in the first half of the name. However, it saves the first half of the name on the Kludge List (so that Dekludge can put it back on demand) and on the SYTCH List (so that it will be put back if the statement fails).

It knows where the first old entry is because it receives a pointer to it. It finds the others by searching the Symbol Table for the name, and it keeps doing this until none remain and the search fails. Then it creates the new entry by calling Add Symbol To Symbol Table, which is a sub-entry point to Identifier Scan (it is not necessary to scan the identifier; we already have it; it is only necessary to make the new entry for it). Then we call Inherit Type, which gets its old type from the old symbol table entry pointer that was given to this routine in the first place.

3.76.1.2 Dekludge DEKLUDGE

This routine undoes the damage perpetrated by Create New ID Entry (above). It is called when the new entry is no longer needed, and we want to restore the old one(s). It steps through the Kludge List (from bottom to top). For each old entry, there will be two items on the Kludge List: a pointer to the old entry in the Symbol Table, and the first half of the name (that is now

clobbered in the Symbol Table). It puts back the first half of the name, so that it will now be recognized by Symbol Table searches, and will take precedence over the new entry because it appears first and is therefore found first. It also searches for the new entry corresponding to each old entry (using a Symbol Table search on the full name). When it finds it (and it must), it calls Inherit Type to give the old entry any new type that the new entry may have acquired. For example, consider the following statements:

```
COMMON Y(10)

COMPLEX Z(5,5)

F(X,Y,Z) = X + Y + REAL(Z)
```

When CREATEID started to produce new entries for X, Y, and Z, one of them, Z, already had a type, so this was inherited to the dummy, Z. Since they both have the same type, it is perfectly legal (though redundant) for Dekludge to inherit the same type back. Y was known to be an array, but had no type as yet. However, by appearing in the statement function, it acquired implicit type REAL, which is then inherited back to the array Y. (Thus Y could no longer be declared any other type.) Not much was known about X, since it had not appeared. The statement function causes an "old" entry to be created, with UNUSED class and UNKNOWN type. Then, the dummy acquires implicit REAL type, and this is inherited back to the UNUSED identifier.

3.76.2 Statement Function Gen ASFDSTAG _ Pass 2)

As far as code generation is concerned, Statement Function Definition acts like three statements in one:

- a FUNCTION statement
- an assignment statement, assigning a value to the FUNCTION name
- and a RETURN statement

Before getting down to business, ASFDSTAG calls REGSRCLN, if DEBUGTRG is set, to register the source line number and associated label on the appropriate lists (3.19). It also generates a branch around the Statement Function code. Now ASFDSTAG acts like it was working on a FUNCTION statement: it creates a Private Temp to be used as the link address temp, and places its pointer in SPLINKPT, saving the old contents thereof (since this Statement Function Definition may be embedded in a subprogram). Then it calls Sprog Head Gen (SPHEADGN, see 3.78.3) to generate the subprogram heading code. In preparation for generating the value assignment code, we call ASMCODE to move all the subprogram heading code from CODELIST to OUTLIST so that any "retroactive subscript" evaluation code (see 2.14.4) necessitated by the assignment code will be properly placed ahead of that code only, rather than

ahead of the entire subroutine code. Now we bring in the Statement Function's expression pointer and invoke the appropriate expression evaluator: EVALOGEX for logical, GTACUREM and MODCVGEN for an arithmetic expression. Having thus guaranteed that the expression's value is in the accumulator by the appropriate mode, we pick up the link temp pointer (restoring the old contents of SPLINKPT at the same time) and give to ASFRETGN, which generates a RETURN. Finally, we define the branch-around label, protect the temps (via PROTEMPS, 2.19.5), and exit.

3.77 STOP Statement

3.77.1 STOP Statement STOPSTA _ Pass 1)

Call Integer or Zero (INTORZ, under PAUSE Statement). This routine was originally written to return a pointer to an Integer constant, if there was one present, or else to the value zero. It was later modified (for IBM compatibility) to also accept an alphanumeric string.

File STOPNO, followed by a pointer to the constant. Exit through JUMPEXIT.

Transmits two items to Pass 2:

1. STOPNO
2. Constant to be displayed (pointer to Integer Constant List or String Constant List)

3.77.2 STOP Statement Gen STOPSTAG _ Pass 2)

There are two varieties of STOP statement, as seen both from the source language viewpoint and the object code. The more traditional form has an integer after the word STOP; the alternate form has a string constant--a message enclosed in quotes.

If an integer appears, Pass 1 has registered it as an Integer constant, and Pass 2detects this by SOTing for ICONSCOD. If it is an Integer constant, we generate a load of the value into register 9, followed by a

```
BAL,6    9STOP
```

The load into register 9 is usually immediate (i.e., if the number is less than 524288).

If a message appears rather than an integer, the location of the string constant is loaded into register 7, and then a

```
BAL,6     9STOPEM
```

is generated.

3.78 SUBROUTINE Statement

3.78.1 SUBROUTINE STATEMENT SUBRSTA _ Pass 1)

LOCAL SUBROUTINEs branch into this same section of code, immediately following the instruction that turns off the Local Trigger. That is the only difference here; the LOCALTRG is tested by FUNCOM3, which we will go to shortly to do the common work for SUBROUTINEs and FUNCTIONs.

First, call File RETURN, which checks to see if this is an internal SUBROUTINE, and if so, whether it ought to put out a RETURN statement to keep people from falling into a receiving sequence. This could be done in common with FUNCTION, except that it has to come before the FAD of SUBRNO or FUNCTNO. Then, since SUBROUTINE is almost the same, but not quite, as FUNCTION, set the SUBTRIG so that the common code will know which it is working with. A thing that is not tested (that could be, with this trigger) is whether to create a scalar with the name of a SUBROUTINE. We have to do it for FUNCTIONs (e.g., in FUNCTION ALPHA there is a scalar named ALPHA that is used to return the result), so we do it also for SUBROUTINEs. That way, if the user should reference such a scalar, he will not get an error. (He will not return any results either, however because we do not pick up the scalar value before returning from a SUBROUTINE.) See FUNCTION Common 3 in section 3.52.1.1 for further details.

Four items are transmitted to Pass 2 (not counting the RETURN that may be created by File RETURN prior to the SUBROUTINE information):

1. SUBRNO
2. Pointer to superfluous scalar
3. Pointer to subprogram name
4. Dummy group plex pointer

3.78.2 SUBROUTINE Statement Gen SUBSTATG _ Pass 2)

SUBSTATG resets the Function Trigger (FUNCTRIG) so that RETURN statements will not attempt to return a value. Then, a decision is made as to whether this is to be a re-entrant subroutine (see also 3.12). The Real-Time Subprogram Trigger (RTSPGTRG) is set if and only if (a) the Real-Time (RT) option was specified, (b) we are not compiling a main program (i.e., MAINPTRG is reset),

and (c) this is the first subprogram definition in the program (i.e., RTTEMPCT, Real-Time Temp Count, contains zero). Next, SUBSTATG brings in (from the Input List) the Sprog Name Variable pointer; this is not used in generating code for the SUBROUTINE Statement or its associated RETURNs, but it is used for FUNCTIONs, the definitions of which are also handled by most of SUBSTATG (see also 3.52.2). The Sprog Name Label pointer is not brought in yet; that will be done later by Sprog Head Gen (SPHEADGN, see 3.78.3). Now SUBSTATG obtains the Maximum Asterisk Dummy count associated with this subprogram. There is one such count for each SUBROUTINE and FUNCTION in the program, and they appear on the Asterisk List (ASTLIST) in the same order as the appearnace of the corresponding SUBROUTINE and FUNCTION statements. For a given subprogram, this count reflects the greatest number of asteriks dummies used in the subprogram definition and all of its ENTRY statements. SUBSTATG pulls it off the top of ASTLIST and stores it in ASTCOUNT for use by SUBSTATG in allocating the Asterisk Return Vector (see 3.78.3.2), and later by Asinine RETURN Statement Gen (ASRETSTG, see 3.73.3). Now SUBSTATG obtains a Private Temp to be used to hold the link address; it then adds ASTCOUNT to PTEMPCNT in order to reserve the n immediately following Private Temps for use as the Asterisk RETURN Vector. Addresses of individual words in this vector will be placed in the appropriate receiving sequence words by Receiving Sequence Gen (RSEQGEN; see 3.78.3.2). The receiving sequence for a subroutine having an ASTCOUNT of three would appear as in the following example:

```
            SUBROUTINE FAROUT (A,*,*,B,*)

FAROUT      LI,5     5
            BAL,6        9SETUPN
            SNGL     A           1st dummy, 'A'
            PZE          2PTMP       2nd dummy, '*'
            PZE          3PTMP       3rd dummy, '*'
            SNGL     B           4th dummy, 'B'
            PZE          4PTMP       5th dummy, '*'
            STW,15       1PTMP       store link address
            ...
            ...
1PTMP       RES          1           temp for link address
2PTMP       RES          1           asterisk return vector, 1st '*'
```

3PTMP	RES	1	asterisk return vector, 2nd '*'
4PTMP	RES	1	asterisk return vector, 3rd '*'

Now SUBSTATG gets down the actual business of generating the receiving sequence and prologue code -- this is all done in Sprog Head Gen (SPHEADGN, see the following section). The only additional thing done explicitly in SUBSTATG is the registration (on ENTRYLST) of the subprogram name, in case the DEBUG option was specified (see 3.19).

3.78.3 Sprog Head Gen SPHEADGN _ Pass 2)

Subprogram heading code is composed of three parts:

(1) special setup code for Real-Time (re-entrant) subroutines and subroutines compiled under the DEBUG option;

(2) the receiving sequence: a call on a setup routine, followed by one word for each dummy;

(3) prologue code: the code required to evaluate and store the subscript multipliers and offsets of all dummy arrays.

SPHEADGN first calls Protect Temps (PROTEMPS) to isolate the ordinary temps used outside the subprogram form those to be used inside. It then brings in the Sprog Name Label pointer from the Input List and generates a Label Definition item for it. Next, it calls Clobber Index Regs (CLOBINDX) so that previously established X-reg context will not be depended upon in generating subprogram code. Now the first part of the subprogram heading code is generated: if RTSPGTRG is set, SPHEADGN calls Real-Time Sprog Head Gen (RTSPHDGN) to generate the re-entrance testing code (see 3.78.3.1); if DEBUGTRG is set, Debug Sprog Head Gen (DBSPHDGN, see 3.78.3.1) is called to generate the subroutine check-in code.

To generate the receiving sequence, SPHEADGN brings in the Dummy Group Plex pointer and PULs the group to the Arg List. How it behaves now depends on whether the plex was an Ordinary Dummy Group Plex or a Multiple Dummy Group Plex (MDMYGRPX). If it was ordinary with n dummies, we gen

```
LI,5    n

BAL,6   9SETUPN
```

or, for n=0, 1, or 2,

```
BAL,6   9SETUP0 or 9SETUP1 or 9SETUP2
```

We then call Receiving Sequence Gen (RSEQGEN, see 3.78.3.2) to generate the dummy list and the STW,15 linktemp

If it was a Multiple Dummy Group Plex with n dummies (the last of which is the Multiple Dummy itself), then there are n+1 words on the Arg List: n-1 fixed dummies, the multiple dummy, and its count variable. For instance, in the statement

```
SUBROUTINE FOOEY (A,B,/W(N)/)
```

the Multiple Dummy Group contains the four entities A, B, W, and N. SPHEADGN pulls off the count variable and saves it in NARGSTMP; it then gens

```
LI,5     -n

BAL,6    9SETUPV
```

and calls RSEQGEN to gen the n-word dummy list and

```
STW,15   linktemp
```

Then SPHEADGN gens the c ode to store the arg count (left in register 4 by 9SETUPV) in the count variable; this involves a mode conversion if that variable is not integer.

Having finished the generation of the receiving sequence, SPHEADGN now falls into Generate Offsets (GENOFFST, see 3.78.3.3) to produce the prologue code.

3.78.3.1 Special Sprog Setup Code Gen RTSPHDGN, DBSPHDGN _ Pass 2 _

When a subprogram is to be re-entrant, SPHEADGN calls Real-Time Sprog Head Gen (RTSPHDGN) to generate the special re-entrance checking code:

```
         LCH,4    1RTMP

         BLEX 1G

         LI,4 1RTMP

         BAL,2    9RSAVE

1G       STH,4    1RTMP
```

See "Real-Time", section 3.12 for a discussion of what this code accomplishes.

When a subprogram is compiled under the DEBUG option, SPHEADGN call Debug Sprog Head Gen (DBSPHDGN) to generate the entry point check-in code:

```
         BAL,6    9DBENTER

         PZE      debug table loc
```

TEXT 'name....'

3.78.3.2 Receiving Sequence Gen RSEQGEN _ Pass 2 _

This subroutine genrates the receiving sequence dummy list, one word per dummy, and the

```
STW,15    linktemp
```

In so doing, it does the appropriate substitutions of asterisk return vector locations for asterisk dummies, and makes a list (on TERMLIST) of pointers to all the array dummies for the convenience of GENOFFST (see the following section). It starts by initializing ASTPOINT (the running Asterisk Return Vector address) to equal SPLINKPT, the link temp address pointer. Then, for each dummy pointer (obtained from the Arg List), if it is an asterisk (determined by SOT ASTPNTR), we increment ASTPOINT by one and gen a PZE to it; if it is an array, we put a copy of it on TERMLIST. For non-asterisk dummies we gen the appropriate argument type word (INTG, SNGL, etc.) containing the dummy's address. When the mode is known, the subroutine Derive Calling Sequence Op Code (DCALSEQ) is called to provide the argument word with the proper op code and "protection" bit; when the mode is unknown (SOM UNKNOWN), we simply use "EVRY".

3.78.3.3 Generate Array Offsets GENOFFST _ Pass 2 _

This subroutine, as the tail end of Sprog Head Gen, generates the code needed to (1) offset the dummy array addresses so that they point to the "zeroth" elements of the arrays, and (2) evaluate dimension and size information for adjustable dummy arrays.

For example, the statement

```
SUBROUTINE FINDJ (A,B)
```

under the declaration

```
DIMENSION A(10,10),B(M:N,K:L)
```

would be compiled as follows:

```
FINDJ     BAL,6     9SETUP    arg/dummy correspondence setup routine
```

```
SNGL A          1st dummy, array A
SNGL B          2nd dummy, array B
STW,15   1PTMP  save link address
LI,9 -11        offset array A
AWM,9    A
LW,9 N          compute second subscript
SW,9 M                multiplier for B; it is
AI,9 1                N-M+1
STW,9    2PTMP  store in temp
LW,9 K          compute the offset for B
MW,9 2PTMP            = (N-M+1)*K+M
AW,9 M
STW,9    3PTMP  store offset in temp
LCW,9    9      use it to offset the array adr
AWM,9    B
```

From B's declaration, it can be seen that the location of B(I,J) exceeds that of B(1,1) by

$$(I-M) + (N-M+1)*(J-K)$$

$$= I + (N-M+1)*J - ((N-M+1)*K+M)$$

Thus the subscript multipliers are 1 and (N-M+1) and the array offset is (N-M+1)*K+M. Each dummy array has in its symbol table entry pointers leading tb the information needed in this process:

- a Group (on the Group List) containing pointers to all of the nontrivial subscript multiplier expressions and the Private Temps into which their values are to be stored; and
- a Group containing the plexes needed to evaluate these expressions.

The offset expression is also available; the same plexes are used in evaluating this. One other expression may need to be evaluated: the size of an adjustable dummy array which is used unsubscripted as an I/O list item (its IOLUSA trait is set). In Arrays, 3.2, there is a diagram showing how all the above information is arranged).

A list of the dummy arrays has been left for us on TERMLIST by RSEQGEN. We take the array pointers from it, one by one, doing the following for each:

(1) Establish in ESIZETMP the element size corresponding to the array's mode;

(2) Get the Dummy Messy Expressions Group (the list of tricky subscript multiplier expressions and the temps into which their values are to be stored) and copy it to DIMLIST; this group is pointed to by rung 3 of the array's ARRAYLST item (see the array information structure table in "Arrays", section 3.2);

(3) Restore the Plex List the way it was when the array information was set up; this is done by copying the Plex Group (pointed to by rung 4 of the ARRAYLST item) to PLEXLIST:

(4) For each subscript multiplier expression/temp pair on DIMLIST (see item 2) we generate (via GTACUINT) the evaluation/store code;

(5) Now we look at the array's offset pointer (rung 0 in ARRAYLST): if it is constant (say, k) and the element size is s, then we gen code to add -sk to the dummy array address; if it is not constant, then we gen code to evaluate it, store it in the Offset Temp (given in rung 5 of ARRAYLST item), multiply it by -s and add that to the address;

(6) Finally, if the array's IOLUSA trait is set, then the array has been used unsubscripted as an I/O List item and we will therefore have to know the array's size in elements; if the array is adjustable, we now gen code to compute its size (its expression pointer is in rung 1 of the ARRAYLST item) and store it in a Private Integer Temp, whose pointer we put back in rung 1 in place of the size expression pointer.

3.79 TYPE Statement TYPESTA _ Pass 1 only)

File the TYPENO (same as WRBCDNO), pick up in WO a pointer to the typewriter output unit number (102), and go to FORTRAN II Output Scan (Section 3.81.1).

3.80 Type Statements TYPESTAS _ Pass 1 only)

The actual quote scans for all the types are done in subroutines, because they are used not only by the type statements, but also by Type Quote Scan (TYPEQS), which is used by IMPLICIT and LOCAL. These subroutines also have to look for trailing "*n", IBM's unfortunate choice for indicating byte size.

Each subroutine looks for a particular type quote and returns an answer whether it found one. If it did, it places the appropriate Mode Word in P0. Actually, this might not be the same type it was looking for if an *n is present or if ADP is in effect. The subroutines are:

1. QSA For Integer (QSAINTG)

   Only returns INTEGER. *4 is ignored. *1 and *2 produce a diagnostic and are then ignored. The only reason *1 is allowed is to common up the code with LOGICAL.

2. QSA For Real (QSAREAL)

   Usually returns REAL mode. *4 is ignored. *8 determines DOUBLE mode, as does Automatic Double Precision.

3. QSA For Double Precision (QSADOUB)

   Only returns DOUBLE. No *n allowed.

4. QSA For Complex (QSACMPX)

   Usually returns COMPLEX mode. *8 is ignored. *16 determines KOMPLEX, as does ADP.

5. QSA For Double Complex (QSAKMPX)

   Only returns KOMPLEX mode. No *n allowed.

6. QSA For Logical (QSALOGL)

   Only returns LOGICAL mode. *4 is ignored. *1 and *2 produce a diagnostic and are then ignored.

Even when a statement begins with a type quote, it may not be a type statement; it may be a FUNCTION statement. So we test for "F" (Function) or"L" (Local); if so, see Type Functions, below (3.80.2).

In addition to processing type and dimension information, this statement has to act like a DATA statement. We do this by reserving the Work List and piling up two-word DATA Pairs (just as the DATA statement itself does). Then, at the end of the statement, we go off and expand them (just as if this had been a DATA statement).

If there is an *n after a name, the type may change. (See Secondary Asterisk Scan Table ASTSCAN, below.)

Now register the type. Then check for dimensions. (See Allow Dimensions ALLOWDIM, 3.37.)

If the next character is slash, indicating DATA, we have to make sure the identifier is a variable of some sort. If it can be a scalar (see Permissibility, 2.8.1), make it one (it may already be, which is valid). Otherwise, it must be an array and it must not be a dummy or in blank COMMON. (See DATA Variable Item Scan (DVARSCAN), 3.34.1.3.) Then call DATA Constant List Scan (DCONLSCN), 3.34.1.6, and put on the Work List both the current identifier pointer and the pointer to the DATA Constant List Plex.

At the end of the statement, if there were any DATA constant list, go to DATA Finish (DATAFIN). Otherwise, use Declarative Exit 2 (legal in BLOCK DATA). This statement transmits information to Pass 2 (on the Output List) only if there were DATA constants (see DATA Statement). It also can produce type and array information.

3.80.1 Secondary Asterisk Scan Table (ASTSCAN)

A secondary asterisk is one that follows an identifier rather than a type word; e.g.

```
REAL FUNCTION ALPHA*8(X)

REAL*4 ALPHA*8
```

Given the current type of an identifier, this table directs you to the place to scan for an *n that is legal. It is used (called) with a BAM (Execute By Mode), returns with the new type in P0 (may not have changed), and always with answer TRUE (because it is using code that, for other purposes, returns an answer).

3.80.2 Type Functions (LOCTYCOM)

A type quote has been found, followed by F or L. Set the Fail Exit (using FEX) to come back if it turns out that this is not a FUNCTION statement. Then check for "LCOAL" (and remember whether there was one) and "FUNCTION" (there must be one) and go to Function Common 1 (FUNCOM1, 3.52.1).

If FUNCTION does fail, control will be returned here with everything the way it was at the beginning of Statement Scan. So we have to go scan the type quote again and pick up the type in P0. (TYPEQS cannot fail, because we have already determined that there was a type quote there.) Also set the Fail Exit back to go to Assignment Statement if Type Statement fails (e.g., a statement such as REALF = 3.)

3.81 WRITE Statements

3.81.1 WRITE Statements (WRITSTAS _ Pass 1)

If the first character after "WRITE" is not a left parenthesis, go to FORTRAN II WRITE Statements. If it is a left parenthesis, signal that this is a write statement (as opposed to a read stateme nt), and go to READ-WRITE Scan, which processes all BCD and binary READ and WRITE statements.

3.81.1.1 FORTRAN II WRITE Statements F2WRSTAS _ Pass 1 only _

This section is much like FORTRAN II READ Statements, except there is no FORTRAN II "WRITE" statement (e.g. WRITE 3, X), so there is no ambiguity and no need to set up fail exits. Each of the following statements looks for its particular quote following the word "WRITE".

3.81.1.2 WRITE OUTPUT TAPE Statement Pass 1 only _

File the WROUTPNO (same as WRBCDNO). Then call BCD Tape Scan, which files a unit number and a FORMAT reference. If an I/O list follows, go to I/O list; otherwise go to End of I/O List.

See FORTRAN II I/O Scan (3.70.1.2.6) for a description of the items that are transmitted to Pass 2.

3.81.1.3 WRITE TAPE Statement Pass 1 only _

File the WRTAPNO (same as WRBINNO). Then call Binary Tape Scan, which files a unit number. If an I/O list follows, go to I/O List; otherwise, go to End of I/O List.

See FORTRAN II I/O Scan (3.70.1.2.6) for a description of the items that are transmitted to Pass 2.

3.81.1.4 WRITE DRUM Statement Pass 1 __

File WRDRMNO. Then call Drum Scan, which files a unit number and a drum address. There must be an I/O list on a WRITE DRUM statement, so go directly to I/O List.

Three items are transmitted to Pass 2 (not including the I/O list items):

1. WRDRMNO
2. Drum number (pointer to expression)
3. Drum address (pointer to expression)

3.81.1.5 WRITE DISC Statement Pass 1 _

Allow alternate spellings "DISC" or "DISK". ("DISC" is preferred by Xerox.) File WRDSCNO. Then call Disc Scan, which files a unit number aand a disc address. There must be an I/O list on a WRITE DISC statement, so go directly to I/O List.

Three items are transmitted to Pass 2 (not including the I/O list items):

1. WRDSCNO
2. Disc number (pointer to expression)
3. Disc address (pointer to expression)

3.81.2 BCD WRITE Statement Gen BCDWRITG _ Pass 2)

This is a pretty easy statement to generate. We have a unit number and a format to handle and a library routine to call. With the unit number we do a little cleverness, and in case it is 108 we save one word of generated code by calling 9PRINT instead of 9BCDWRIT. (9BCDWRIT expects the unit number in register 9, whereas 9PRINT knows that unit 108 is intended).

We JRS to NXINWORD for the unit number, then QJS to CHEKUNIT which, among toher things (see 2.23), checks to see if it is 108; if it is not 108, we guarantee it integer in AC (register 9). We then call FORMGEN (Format Gen) which calls NXINWORD for the FORMAT pointer, then generates appropriate code to get its address into register 7. Then we generate a BAL,6 to either 9BCDWRIT or 9PRINT. Then we Reserve the Delayed Code List (DCODLIST) (see 3.60.2.3 to find out why) and exit through Statement Exit (STATEXIT).

3.81.3 Binary WRITE Statement Gen BINWRITG _ Pass 2)

This statement is handled in the same way as the Binary READ Statement except that the library link generated is to 9BINWRIT rather than 9BINREAD. See the writeup for Binary READ Statement Gen (3.70.3).

3.81.4 WRITE DRUM Statement Gen WRDRUMSG _ Pass 2)

WRITE DRUM is just like, say, Binary WRITE, except that in effect it has two unit numbers instead of one. These are called the Unit Number and the Address and are sent to the library routine in registers 9 and 11, respectively.

We ADR the 9WRDRUM Link and branch to READ/WRITE DRUM/DISC (RWDRMDSC, 3.70.7) which does all the work.

3.81.5 WRITE DISC Statement Gen WRDISKSG _ Pass 2)

WRITE DISC is just like, say, Binary WRITE, except that in effect it has two unit numbers instead of one. These are called the Unit Number and the Address and are sent to the library routine in registers 9 and 11. respectively.

WE ADR the 9WRDISC Link and fall into READ/WRITE DRUM/DISC (RWDRMDSC, 3.70.7) which does all the work.

CHAPTER 4. EXTERNAL MECHANICS

External mechanics that have nothing in particular to do with compiling the FORTRAN language.

4.1 Theory of Monitor Interface

When writing a compiler, it would be convenient if the hardware instruction set provided just the facilities you needed, such as dynamic list searching, recursive calls, character scanning, etc. By using an interpretive language, we have essentially proceeded as if the hardware did provide these facilities. The compiler is written in a specially designed instruction set (POPs), and the interpreter provides the facilities.

Similarly, it would be convenient if the hardware provided all of the input/output facilities desired. There have been machines that came close; on the Univac I, you could read a block of data off a magnetic tape (complete with error checking and rereading if necessary) with one instruction. Since the hardware does not directly provide all the facilities needed (and there is certainly good reason for it not to do so), Monitors are written to interface with the hardware. Often however, the Monitors do not provide all of the facilities needed, and the facilities that are provided vary considerably from Monitor to Monitor in how they are handled. In order to centralize the problems associated with Monitor interface, we have gathered it all into one assembly. The rest of the compiler is written as if there were a Monitor (or hardware system) out there that did exactly what the compiler wants. The one assembly that interfaces with the Monitors is called MERRIMAC.

MERRIMAC is designed to provide simple handles to the rest of the compiler. Calling sequences are short, and MERRIMAC takes care of as many of the details as possible. The rest of the compiler does not need to worry about DCBs, FPTs, JITs, Opens, Closes, traps, or abnormal returns. A typical request on MERRIMAC would be "Read an 80-character line from the source input file, and put it here. And don't come back until you do."

In theory, changing Monitors only involves changing MERRIMAC. For the most part this is true, but there are exceptions. For example, one of the MERRIMAC routines is Read One Character From OC. Some of the Monitors simply do not provide the facility to read one character from OC, and no amount of finagling by MERRIMAC is going to provide it. In this case, the calling section has to be changed to do something other than read one character from OC. As another example, the object listing and summary sections of the compiler were designed to print a listing up to 108 characters wide, which would fit in eleven inches. When operating in a Teletype environment, where the paper width is much less, it would certainly be possible for MERRIMAC to fulfill the exact wording of the request and get those 108 characters out somehow (like on two lines), but this is ugly, and it is preferable for the listing sections to know about limited width and make their listings narrower. In most cases, however, MERRIMAC does do the finagling necessary to provide the compiler what it needs, one way or another.

For the purposes of this document, the operations that MERRIMAC performs are divided into three categories. The first (4.2) is normal Input/Output, and the second (4.3) is Miscellaneous. These two categories consist of operations that the compiler knows it needs. Obviously you have to be able to read souce lines, write scratch files, get to top of page on listings, abort to the

Monitor if a serious error occurs, etc. The third category (4.4) consists of Special Operations that are necessary only because of the environment in which the compiler finds itself. Each of the operations in these three categories will be discussed, along with a description of the routine in MERRIMAC that performs it.

Finally, there is another area (4.5) consisting of "Other Considerations" that affect the design of the compiler, but are not exactly "operations" performed by MERRIMAC. Sometimes, in fact, they result in non-operations. For example, since the binary object language does not provide a facility for specifying that a COMMON block must begin at an odd location, there is obviously no operation performed to use such a facility. But there could have been (originally there was), and therefore such a consideration is important to a full understanding of the compiler. Other examples are error severity conventions, where to output REFs and DEFs, what protection type to generate, and what size of unit numbers to allow.

There is also, of course, a whole set of similar considerations in effect in the Run-time, but those are not documented in this manual. Nor is the free-standing MERRIMAC documented. It provided all the same services as the Monitor-oriented MERRIMAC, but dealt with the hardware directly on a completely dedicated computer.

4.2 Input/Output Operations

4.2.1 Read a source line from SI (SIREAD)

This is essentially the only inputting done by the compiler, and it is called from only one place, Next Input Character (NXTINPCH in PASS1ML), when the character scanning routines need another source card (see Character Scanning, 2.2). The other two input routines, DIREAD and DCREADCH, are used only by the compiler debug package, which is not a standard mode of operation.

SIREAD is called with two arguments: the byte address of the area to read into and the size (in bytes) of the record. Actually, this second argument should not be passed at all, since it is always the same; SIREAD could set that up.

READ and DOREAD are described below. SIREAD has to do a little maneuvering before it can call READ.

First we have to initialize the continuation trigger (CONTTRIG) to indicate that continuation may occur unless later events serve to inhibit it. Unfortunately, this trigger is defined in such a way that when it is TRUE (negative), it means that continuation cannot occur. It probably should have been called the "no-continuation" trigger, except that there already was such a thing (NOCONTRG), which is used, not to say whether continuation is allowed, but whether it has occurred. All of this is described under Card And Line Handling, Section 2.3.

Then we have to perform a task which we would prefer to see handled automatically by the Monitor. At the end of a compilation, when no BC option has been specified, we have to read the C device to see if there is a new control card. If SI and C are the same, there may not be another control card; there may be the first card of another source program. This kind of consecutive compilation is allowed (see Interpret Control Card Options, 4.3.3). Unfortunately, this means that the record we just read from C is really an SI record and we did not want it until the first call on SIREAD, which will come along later, but we have it and we cannot very well give it back and say, "Here, Monitor, give me this again when I ask for SI." (Though something like this is probably what we would like to see happen.) What we do instead is set the SI From C Trigger (SICTRIG), and if SIREAD finds this trigger set, it does not actually do a read from SI; it picks up the information in the C buffer and moves it over to the place requested in the call on SIREAD. Then it turns off the SICTRIG.

Then we initialize the EODCOUNT to two. This will be used by DOREAD to determine what to do with EOFs and EODs between source programs, based on the setting of the BC option. Finally, we fall into READ.

4.2.1.1 READ

READ calls DOREAD immediately, to perform the actual I/O operation. All the finagling comes afterwards, beginning with: if this was a read from SI, turn off the FIRSTSI trigger. This trigger is used by DOREAD (in conjunction with EODCOUNT and the BC option) to decide what to do with EOFs and EODs.

Next we massage the input record in four ways.

1. Look at the last character in the record, and if it is a new line or carriage return, change it to a blank. We wish the Monitor would assume responsibility for this. When we ask for a record, we mean the actual useful data in it, and from a typewriter, the N/L is not part of the useful data; it is a control character that indicates end of line, and as such it could be removed by the Monitor, and the size of record that it reports could be lowered by one. Another problem with this procedure is that it is currently done regardless of the input medium. This means that a card or magnetic tape source line cannot have a carriage return character in a Hollerith field as the last character in the record. This is not likely on a card, since it would have to be in column 80 (unless there is such a thing as short cards), but it could happen on short magnetic tape records, and we always tried to be completely clean about allowing people to use strange characters on devices where there is no reason not to.

2. If we are on-line, and if the current READ is from TY or OC, look to see if the character immediately preceding the N/L is a colon. If not, then there cannot be a continuation line following this one. (This is a UTS convention trying to do something about the horrible way that the original FORTRAN designers specified continuation cards, i.e., you do not know if there is one

until you read it). This is signalled by setting the CONTTRIG TRUE (negative). If there is a colon, change it to a blank and leave the CONTTRIG FALSE, meaning there can be (but does not have to be) continuation.

3. Look for a tab character in columns 1-6. If there is one, move all the characters following it so that they begin in column 7. If this causes characters at the right end to extend beyond the maximum size of the input area (as specified in the call on SIREAD or whatever), they just get lost. The gap opened up where the tab character is filled with blanks. Originally this code was even more complicated. We had to look in a Monitor table to see where the user had specified his tab stops, and move the information accordingly. Now it is coded to automatically assume that the only stop is in column 7. Neither of these is completely desirable. Although it is true that the default tabs for FORTRAN ought to be in 7 and 73, the user ought to be able to change or add to these if he wants. And since it is a Monitor facility that he uses to do this, the Monitor could do the tab processing for everybody, and hand over records with the characters properly moved around and blanks inserted. There are rumors that this may actually have been done, and that all this code is superfluous.

4. If the actual record size read was smaller than that requested, fill in the remaining positions with blanks.

4.2.1.2 DOREAD

This routine performs the M:READ on the requested file. If an end-of-file is encountered, or if the current file is the same as C and a control card is encountered, or if some other abnormal condition results, this is handled by the READ Abnormal Return, described below. There is one case, however, in which the Monitor does not think the situation is abnormal, but we do. If the current file is SI and it is not the same as C, and a record saying "!EOD" is encountered, that is really the same as an EOF (or EOD), so we go down and handle it as such. Otherwise we just exit.

If an abnormal return occurs from the M:READ, we are mainly interested in the end-of-file conditions, and these are a little confusing due to inconsistencies of terminology within the Monitors and between the Monitors and the FORTRAN language. In FORTRAN, an end-of-file is merely a special mark that indicates the end of one file of data; other files may follow, and the "END=" and EOFSET features are provided to process multiple files. In the Monitors, however, end-of-file tends to be a stronger "end", indicating that there is no more data. The EOF return occurs when a new control card is encountered (at which point there cannot be any more data). The weaker "separator" kind of end-of-file that FORTRAN wants to deal with is generally called an EOD (End Of Data). Thus a physical end-of-file (on an unlabeled tape) gives an EOD abnormal return, but the procedure used to write a physical end-of-file is M:WEOF (there is no M:WOD). On labeled tapes, M:WEOF does nothing, because labeled tapes do not really support the concept of multiple files as defined in FORTRAN. However, this is more of a run-time

consideration. The important fact to the compiler is that it must be prepared to deal both with EOF and EOD returns, as described below.

The treatment of EOFs and EODs applies only to the SI file (and depends on the setting of the BC option). So we first check to see if this is M:SI we are working with. If not, this condition is an error and is discussed later. Remember also that an !EOD record when SI is not equal to C is treated exactly like an EOD abnormal return.

If the current read is on SI, but the abnormal return is not EOD or EOF, this is also treated as an error, and is handled below. The treatment of EOD and EOF from SI is based on the following rules:

1. A physical end-of-file (tape mark) on an unlabeled tape produces an EOD condition.

2. An !EOD record on a card reader, typewriter, or paper tape reader also produces an EOD condition.

3. An !EOD record from any other source (when SI and C are not the same) does not produce an EOD condition (it does not produce any abnormal condition), and we think it should, so we make a special check for it (as described above) and call it EOD.

4. Any control card (beginning with !), when SI and C are the same, produces an EOF condition.

If this is <u>not</u> the first read from SI (as determined by the FIRSTSI trigger, which is set up in FOPTIONS, the section that interprets control card options), then we treat EOD and EOF the same. What we do is to insert an exclamation point into the first character, so that the compiler can see that this is a special line that really should not be there. What the compiler then does (see Label Field Scan, 2.9.5) is to treat the line as an indication of end of program. This enables us to finish off the compilation and go through Passes 2 and 3, rather than just aborting in the middle of Pass 1. Note that if a control card appears where there should be a source line, and SI is different from C, this is not detected by anyone except the upper levels of the compiler (in Label Field Scan).

If this <u>is</u> the first record read from SI, and it is an EOF (meaning a control card has been encountered on the C device), we just exit to the Monitor without doing anything. There was no program to compile, and now the user has a new Monitor task to perform. If it is an EOD, on the other hand, (indicating tape mark or !EOD card), this is what FORTRAN really means by an "end-of-file", and we have to take into account the BC option. If no BC option was specified, we ignore the EOD and go on and read another record, and we will keep doing this until we find a program or we hit an EOF (because of a control card). If BC was specified with no count, then we are supposed to terminate on a single EOD, so we exit to the Monitor. If BC was specified with a count (i.e., BC(n)), then we are only supposed to terminate on a double EOD (or, of course, on exhaustion of the count, but that is handled by

FOPTIONS). The double EOD is determined using the EODCOUNT which was set up above by READ. When the first EOD is read (and BC(n) was specified), EODCOUNT is reduced from 2 to 1, and we go back and read again. If a second EOD is read, the count is reduced from 1 to 0, and this causes us to exit to the Monitor.

All other abnormal returns are considered errors, except "lost data", which is ignored. Everyone knows that FORTRAN statements are 80-characters long, and if source records are longer than that, we are not interested in what is beyond column 80. All the other conditions (which means mainly EODs or EOFs on scratch tapes) receive a diagnostic about unexpected end-of-file. If it is a scratch tape that this occurs on, we put out a special message identifying which scratch tape (X1 or X2). This is done by the same section, and with the same message, used for checksum errors on scratch tapes.

4.22 Write a line on LO LOWRITE)

This is used mainly by the PRL POP, which in turn is used in a large number of places for printing the source listing, diagnostics, object listing, and summary. This routine is also used directly by the debug package for all "DP" requests and for keeping the printer log of typewriter operations.

LOWRITE, and the other WRITE routines that are described below, use a routine called WRITE, which uses a routine called DOWRITE. Rather than loading the required DCB into a register before going to WRITE (as the READ routines do), the WRITE routines all fall into one another, and a counter called DCBREQ is incremented at each step. Thus, the higher up you branch into the WRITE routines (the highest is OC), the higher the count will be. (PRNTDOML works independently of this ladder. It just sets DCBREQ to the proper value and calls WRITE.) LOWRITE is lowest on the ladder and, in fact, does nothing at all before falling into WRITE. This is possible because there is much less finagling required to get output operations done the way we want them than input operations.

4.2.2.1 WRITE

First pick up the value of DCBREQ to determine what file we are outputting to. Then reset DCBREQ to zero. This is a kind of post-initialization that would be dangerous in a debugging, or interactive environment, since one culd interrupt processing after DCBREQ had been changed, and then try to branch back to a point that would try to do some output, without DCBREQ having been reinitialized to zero (this is done only at the very beginning of compilation). In a batch mode, this has not been a problem, but under UTS (especially with debug in operation), it might be. Free-standing MERRIMAC did not work this way.

At the WRITE level, there is one Monitor requirement that has to be taken care of. (There will also be one at the DOWRITE level.) If you are on-line, there

is a set of flags called J:OPT that tells, for certain DCBs, whether they are really currently active or not. If the on-line user has not specified a place to put certain output, then it is as if the DCB were assigned to NO, even though that is not what it says. So we have to check first whether we are on-line. If so, we check whether the DCB we are about to use is one which can be "turned off" by the user (LO, GO, and DO can; BO, SO and OC cannot). If so, we check to see whether he has turned it off. And if so, we bypass the call on DOWRITE.

4.2.2.2 DOWRITE

This routine performs the M:WRITE on the requested file. Normally it does so without speicifying any abnormal or error returns. There is nothing we can do about them that would be particularly helpful, so we let the Monitor take care of them. However, in the case of scratch files under RBM, certain abnormal conditions are not properly dealt with, so there is a special routine, DOWRITEX, used only by XWRITE, that specifies an abnormal return and aborts with a diagnostic if it occurs. (See Write A Record On A Scratch File, 4.2.7).

Under all other circumstances, only one special test has to be made before performing the M:WRITE. If this output is going to a typewriter (which can mean the OC Op Label or a typewriter device or a sure console device, and we have to test for all three separately -- see TYTEST 4.4.4.5), we do not want it to type trailing blanks, so we look through the line, starting at the back, and reduce its size to the point of the last non-blank character. We would prefer this to by handled automatically be the Monitor. Any line that it knows it is going to terminate with a carriage return (which it does) will never have to have trailing blanks typed. Of course, lines that are not being terminated with a carriage return should have trailing blanks typed, since they may be used to move the carriage over to a particular place for subsequent operations.

4.2.3 Write a record on GO (GOWRITE)

This is used only by OUTBO (see 2.28) to produce binary output on the GO file. It works exactly the same as LOWRITE (4.2.2).

4.2.4 Write a record on BO (BOWRITE)

This is used only by OUTBO to produce binary output on the BO file. It works the same as LOWRITE and GOWRITE, except for one curious problem. Originally, there was nothing special about BOWRITE; it just did a WRITE, exactly like GOWRITE. Then we discovered that the Monitor's default mode for the BO (BINARY Output) file was BCD, not binary. This meant that all object decks punched out were meaningless, since they were punched out in EBCDIC and not in binary. So we put in code to always set the mode of M:BO to binary. However,

this is a very time-consuming operation, so we also put in code to check and see if the DCB is already binary, in which case we do not have to change it. However, in the F00 release, the Monitor's use of DCBs changed so that the mode bit (used to check for binary/BCD) does not necessarily indiate mode. If M:B0 is not assigned to a DEVICE, the previous mode bit indicates cylindericity rather than mode. Therefore, an additional check occurs before the check for proper mode.

4.2.5 Write a record on SO SOWRITE)

This is used only by Next Input Character (In PASS1ML, see 2.2.3) to copy an input record (from SI) onto the SO file. It works exactly the same as LOWRITE and GOWRITE (4.2.2 and 4.2.3).

4.2.6 Write a line on DO PRNTDOML)

This routine is used when printing diagnostics to make sure that they not only go out on LO, but also on DO (Diagnostic Output) if DO is going to a different place than LO. Obviously if they are being listed on the same file, you do not want the lines to come out twice. (It would be convenient for this feature to be automatically provided by the Monitor.)

PRNTDOML differs from LOWRITE in a couple of ways. First, it is called from POP language with a BRL. This is done after a message has been prepared (by PRERR1 or PRQ POPs or whatever) and before the PRL that outputs the line on LO. In fact, I notice that not all the error messages were caught when this change was put in. Some of the diagnostics in Pass 2 and Pass 3 will not come out on DO. This should be changed.

Also, whereas LOWRITE is called with parameters specifying location and size, PRNTDOML, being called from POP language, is not passed any parameters. It knows that it is supposed to print the Print Buffer (PRBUFFER) using Print Buffer Size (PRBUFSIZ). As with LOWRITE, PRNTDOML calls WRITE to do the actual output.

Finally, of course, PRNTDOML has to determine whether LO and DO are the same before doing anything. Most of the work is done by LOEQFILE (see Test For Correspondence Of Files, 4.4.4), but there is even more. Similar to the situation outlined above, before we can check whether DO equals LO, we have to check to see if it is all right to check.

If you are on-line, there is a set of flags called J:OPT that tells, for each DCB, whether it is really currently active or not. If the user does not sepcify someplace to put LO, then there is no LO, even though it may be assigned to something elsewhere. Thus the testing performed by LOEQFILE may not be enough: it may tell you that LO and DO are the same, but in fact, LO has been turned off, so we had better output on DO or the diagnostics will not come out anywhere. And before we can test that, we have to test whether we are on-line, since otherwise the J:OPT word tells us nothing meaningful.

4.2.7 Write a record on a scratch file XWRITE

This routine is used in only two places: One is Dump Output List (in INTERPTR), which is called when working storage runs out and then periodically thereafter. The other is Output Last Scratch Block ML (in PASSOF), which makes sure that the last (probably incomplete) scratch record gets out.

The value (1 or 2) in OUTUNIT determines whether scratch output is currently going out on M:X1 or M:X2. Since many compilations never use the scratch files, we do not want to rewind them until they are actually needed. So each file has a trigger indicating whether it has been used yet. If not, we call Rewind Scratch File (REWIND, 4.2.9).

The information going out to a scratch file comes from the Output List. The record that we would like to create needs two words on the front, specifying record size and checksum. Rather than transfer all the information to a buffer, preceded by two control words, we simply pop the two control words right into memory ahead of the area to be written out. To make sure we do not clobber anything, we pick up the two words that were there, and then put them back after the write. All this is done with the aid of the two subroutines XSETUP and CHECKSUM. The checksum is computed by simply adding together all the words except the two control words. Similarly, the "size" control word is the number of data words and does not include the two control words.

An early bug that we had with this compiler resulted from this popping the control words ahead of the area to be written out. Since the Output List is the first one (starting at lower memory), the two words preceding the very first record to be written out may lie in someone else's page of memory, and that page may be write-protected. So we have Get List Memory initially allocate a two-word empty space ahead of all the lists. Thus the worst case write (the first one) is guaranteed that it can pick up the two preceding words and then put them back.

We next come to a Monitor consideration that caused us to change our design. Originally, the actual outputting was done by DOWRITE, like everywhere else. We did not specify error or abnormal returns, because there was not much we could do about them, and we were just as happy to let the Monitor take care of them. But it turns out that RBM does not handle them; it just ignores them. If you try to write a scratch file, and there is no more RAD space, or something equally catastrophic, RBM does not print a diagnostic or abort or do something useful. It comes back to you, just as if it had done what you asked, only it has not. Why this error checking was done by adding a whole new DOWRITE (called DOWRITEX), instead of just adding the error and abnormal returns to the existing call I do not know. Perhaps it was so that if an error occurred, we would know for sure that it occurred on a scratch file and could output a correspondingly informative diagnostic.

4.2.8 Read a record from a scratch file (XREAD)

This is used only by Read Next Scratch Tape Record (READTREC, in PASSOF), which in turn is used only by Next Input Word (in PASSOF), which Passes 2 and 3 use to process the information on the scratch files one word at a time, and by the LODI POP, which the same passes use to transfer a number of words from the Input List. If either of those places runs out of data on the Input List, they call XREAD to get more. There is no special provision for not finding any more. The compiler will never request more data back from a scratch file than it put there. If the scratch file comes to end-of-file, that is an error and is diagnosed.

As described under XWRITE, there are two extra words on the front of each scratch file record. These are read into the two cells preceding the actual record area. So we pick up those two words before the read and then put them back again afterwards.

No special "abnormal" checking is needed for RBM, because DOREAD does check for abnormal codes.

If an abnormal end-of-file occurs, or if the checksum does not compare, we put out a special error message indicating which scratch file (1 or 2) the error was on. Then we abort.

4.2.9 Rewind scratch file (REWIND, REWINDML)

REWINDML is called from POP language and is used mainly in the overall control in FORTRAN4. At the end of Passes 1 and 2, we call Finish Output (FINOUTPT), which makes sure the last scratch record is out and then rewinds the appropriate tape. In addition, at the end of Pass 2, we call REWINDML to rewind scratch file 1. It is not needed again; this just gives us a head start in case there is going to be another compilation. Similarly, at the end of Pass 3, we start scratch file 2 rewinding.

REWIND is called in machine language and is used only by REWINDML and by XWRITE, which needs to rewind a scratch tape the first time it is asked to write on it. Note that XREAD does not have to rewind its tapes the first time it is asked to read them, because this has already been done by the call on Finish Output that was done at the end of the previous pass.

In the case of rewinding, it is completely straightforward to do the actual rewind, and it is the same on all the Monitors. If a scratch file has not yet been used (indicated by the two-word table X1USDTRG), REWIND ignores the request to rewind it. Thus, FORTRAN4, can safely go through the necessary procedures, knowing that nothing will happen if it is not appropriate.

4.2.10 Top of page on LO LOPAGE)

This is used in only two places, both of them POPs. One is GNL (Guarantee N Lines on LO Page), which page ejects if there are not enough lines left on the page to fit a table, or a diagnostic and its marker line, or the final statistics. The other is the PRL POP, when used with the operand PAGE. This only occurs in three places: before the source listing (in PASS1), before the object listing (in PASS3), and before the summary (in SUMMARY). See also Paging Conventions, 4.5.4.

It would be preferable to us if the Monitor had a simple procedure for specifying top of page, if this procedure were the same under all the various monitors, and if it would do something reasonable when LO was not assigned to a printer. For example, if LO were assigned to a magnetic tape, then it could do nothing, unless the tape were a listing tape, in which case it could write something that would cause a top of page when the tape was listed. If LO were assigned to a typewriter, it could simply upspace one line, unless the program was operating in an on-line "PLATENS" mode, in which case, it could upspace the appropriate number of lines to get to the next "page". Probably this latter is not an option that should be very common, since it produces a lot of unnecessary paper and wastes time.

Actually, the Monitor does not do what we would like when LO is assigned to an on-line terminal, so we have to check for this (using TYTEST -- see Test For Correspondance Of Files, 4.4.4). Before we can do this check, we have to check whether we are on-line or not.

In addition, three different methods have to be used to effect the top of page, depending on which Monitor you are under. BPM and UTS have an M:DEVICE CAL for this purpose. RBM does not have such a thing, so you have to actually write out a one-character record. But you have to be in VFC (Vertical Format Control) mode for this to work, and this requires an M:DEVICE call. What the current coding does is then unconditionally get out of VFC mode after the M:WRITE. Probably what this code should have been changed to was what the library was changed to, namely, check to see if the user has requested VFC mode already. If he has not, then set up VFC mode. Then do the write. Then check to see if you had to set up VFC mode, and if you did, turn it off. This leaves the mode the way it was when you came in, and is probably preferable.

And finally, it appears that the M:DEVICE CAL (under BPM) for top of page does not actually allow you to start printing at the top of the next page. It upspaces after page ejecting, and therefore starts you off on line two of the page. This can have the amusing effect that shortening the source program can lengthen its listing. Consider the following sequence:

1. Print one source line.
2. Print one source line.
3. Guarantee 3 lines on page to fit following souce line with error.

4. Print source line, error marker, and error message (3 lines).

Suppose that step 2 was printed on the last line of the page. The system automatically ejects to top of page, so the "Guarantee 3 lines" does not have to do anything because it is a top of apge and there are plenty of lines available on this new page. However, if we try to shorten the listing by removing step 1, here is what happens. Step 2 is now printed on the next to last line on the page. Step 3, in order to make room for step 4, now has to issue a page eject, which causes line 1 of the new page to go unused. This means that step 4 gets printed one line beyond where it was printed before, even though the listing is presumably shorter. Had this been known at the time LOPAGE was written, we probably should have tried to code around this problem. Perhaps it can still be done.

4.2.11 Read a debug line from DI DIREAD - used only by F4DEB

This is used only when the "CRD" request has been given to the compiler debug package, i.e. debug input is not coming from the OC (Operator's Console) device. Since there is no such I/O Op label as DI, MERRIMAC interprets this to mean the C device. So all this routine does is set up the file to be M:C, and then go directly to READ, following the same course of action as SIREAD except for all the special cases that apply to input from SI.

4.2.12 Read one character from OC OCREADCH - Used only by F4DEB)

This is the normal routine used by F4DEB to process debug commands, when "TYP" is in effect instead of "CRD". Since the debug package reads one character at a time, until it encounters a period, this routine is supposed to be called with no arguments and to return in a register the one character that it read from M:OC. In fact, however, later monitors have not provided the ability to read only one character from OC (they require that every line end with a N/L), so this routine has been changed (though unfortunately not the comments explaining it) to read a full record. The debug package has also been changed to accept a full record and search for the period. Probably, if debug is ever made truly operational under current monitors, the period terminator should be eliminated, and N/L used instead, unless this causes problems because the terminal paper is not wide enough to hold good-sized debug commands. OCREADCH thus operates now exactly like DIREAD, but from OC.

4.2.13 Write a line on OC OCWRITE Used only by F4DEB)

This is used mainly by DOUTPUT, which does all the typing of debug information. It is also used, however, by the main part of debug, the assembly F4DEB, in two places: one is to output diagnostics when unrecognized debug commands are found, and the other is to output the " " prompt character before accepting OC debug input.

This routine is supposed to put out a carriage return before the record, and no carriage return afterwards. This has the effect that output lines will always begin at the left side of the paper, regardless of where the last use of the typewriter left the carriage positioned. It also enables the prompt character to come out on the same line as the input to be typed in. However, OCWRITE was also designed to interface with an inputting technique that terminated on period, rather than carriage return. All of this had the pleasing effect that both input and output finished with the carriage somewhere across the page, and both input and output began with a carriage return (since input was always preceded by the carriage return and the prompt character). When OCREADCH had to be changed (because it was not possible to read one character at a time and terminate on period), OCWRITE was also changed to have its carriage return at the end instead of at the beginning. This means that: (a) if the carriage happens to be positioned in the middle of the page, an output line may not fit on the page; (b) the prompt character is on a separate line preceding the input line. This whole set of procedures should be re-examined when debug is made to run under UTS or any other console-oriented system.

Currently, OCWRITE works exactly like SOWRITE, GOWRITE, OR LOWRITE, except that it lengthens the record by one character and sticks in a carriage return at the end.

4.3 MISCELLANEOUS OPERATIONS

4.3.1 Get list memory i.e., working storage - GETLMEM)

This is called only once, in the assembly FORTRAN4, when the compiler is first entered (not between consecutive compilations). It sets up the parameters PBOTLMEM and PTOPLMEM to identify the bottom and top of available working storage (for use by Reassign Memory). How it obtains working storage from the Monitor depends on which Monitor it is.

If on-line (UTS), we get one page, using the M:GP Cal. I think this probably should be changed to get, say, four pages initiallly, because 512 words is not really a whole lot for the compiler, and it is probably going to be looking for more shortly.

On BPM, we get all of memory by using M:GP to request 255 pages. The Monitor returns in registers 8 and 9 the number of pages and the starting point of the memory it has given us, and we do not really care how big it is. We just go ahead and use it, and if it is insufficient, we overflow and abort.

On RBM, we also get all of memory, but there is no M:GP Cal, so we get the two values from absolute parameters called BPEND (Background Program End) and BCKEND (Background Memory End). These are actual location values, rather than number of pages, so we do not have to multiply by 512.

Whatever the operating system, we also have to check whether the top of the compiler is located above 64K in memory, since if it is, the INTerpret instruction will not work properly (see 64K Restriction, Section 4.5.6). We do this by looking at the low end of working storage and assuming that this is the same as the high end of the compiler. If any Monitor should decide to put the compiler high and the working storage low, this check would not work. Similarly, if it decided to put the working storage much higher than the end of the compiler, we might produce an unwarranted diagnostic. Note also that the working storage itself is allowed to be above 64K, but not the actual instructions of the compiler. If this test shows a violation, we print a diagnostic and abort.

One last detail: because of a peculiarity in the handling of scratch files, we need to assure a two-word empty space just beneath the beginning of list memory (see XWRITE, 4.2.7). So we add 2 to the lower limit before storing it in PBOTLMEM.

4.3.2 Get additional list memory GETAMEM

This is called when Reassign Memory needs more working storage. If it can get more storage, it increases PBOTLMEM and LASTLIM (the limit of the last list -- see Initialize Lists, 1.4.8) accordingly. If it cannot, it leaves those values alone and returns with zero in register 8 and the condition codes set so that a BEZ will tell whether it got any. On RBM, there is no point in asking for more storage, since we got it all in GETLMEM. On BPM/UTS, we may have gotten all or some of the working storage, but rather than test which we did, we just let the Monitor tell us whether or not there is any more. We always ask for one page at a time.

4.3.3 Interpret control card options FOPTIONS)

This routine is called initially when the compiler first comes into memory, and it determines what options have been specified by the user It is also called subsequently after each compilation to determine whether to go on compiling, using the old options, or to return to the Monitor for another control card. This latter determination is affected by the user's use of the BC (Batch Compile) option, by whether we are on-line, and by what FORTIONS finds when it reads from C.

FOPTIONS can end up doing any one of four things:

1. Interpret options, set up the corresponding compiler triggers, and make a normal return.

2. Do nothing and make a normal return, leaving the options as they were for another compilation.

3. Same as 2, but set the SICTRIG (SI from C), to tell SIREAD that we have already read the first source line.

4. Return with skip, to tell calling routine to exit to Monitor.

The decision as to which of the above to do is made according to the following table:

Some of the above situations merit special comment. "First time", above, means the first time that FOPTIONS is called, which is right after the compiler has been brought into memory. Presumably the Monitor has seen a !FORTRAN control command in order for this to happen. We have to issue a read on C to get at the control command and interpret the options. This read should always give an abnormal return (06), indicating a control command has been read. The other two cases shown above under "first time read from C" probably should never happen.

When it is not the first time, a variety of things happen depending on the BC option. If plain BC is specified, we do nothing in FOPTIONS and just let the compiler go on reading SI until it finds an EOF or EOD condition (see SIREAD). If BC(n) is specified, we read the specified number of programs and then return to the Monitor, since there will presumably be a control card coming along. Note that this has the effect, if SI=C, and there are three programs following a !FORTRAN card with BC(2) specified, that the third program will not be compiled, because the Monitor will read it and skip over it looking for a control command. If there was no BC specified, we go ahead and read from C (unless we are on-line, in which case we never read more than one program unless specifically asked to). This is so that, if SI=C, and there are several source programs following a !FORTRAN card with no BC specified, we will go ahead and compile all the programs. In this case, we read from C and discover that there is not a control card there. This means that it must be the first source card of a new program, so we set the SICTRIG (SI from C), so that when SIREAD is first called (as indicated by the FIRSTSI trigger), it will know that the first source line has already been read, and will pick it up from the buffer instead of actually reading SI.

The above procedure has one peculiar effect. If SI is different from C and the user specifies BC(1), we read one program from SI and then return to the Monitor, which is a sensible procedure. However, if he speifies no BC, we read one program and then read from C ourselves. Normally we get a control card and return to the Monitor, which is all right also. But suppose we get a non-control card. If SI had been the same as C, this would have been fine; this card would be the first card of a new source program. With SI not equal to C however, who knows what this card is? What happens is that we treat it as the first card of a new program, but all the other cards come from the SI device, whereas this one came from the C device. Admittedly, this is not really a legitimate situation; there is supposed to be a control card there. But our reaction to the situation is a little wierd. What we probably ought to do in this case is check to see if SI and C are different, and if so, return to the Monitor rather than reading C. However, given the fact that there is no routine to test for SI=C, and given how cumbersome it is to write such routines (see Test For Correspondance of Files, 4.4.4), it is probably not worth changing.

Once we get to the point where we really do want to read C and get a control command to interpret, if it is the first time, that is fine. If not, we cannot really process the control command; we have to go back to the Monitor and let it bring us into memory again (assuming it finds that the control command is !FORTRAN). If we find a physical end-of-file, or an !EOJ card, we

just ignore it. Note that this does not thwart the testing going on in conjunction with the BC option, because when BC is specified, FOPTIONS does not really do anything; all the reading is done by SIREAD, and that is where the EOFs are handled.

Note also that under RBM, it is not allowed to try to read a control command that the Monitor has already read (whereas under BPM it is mandatory). In this situation, the control command has to be picked up from the s;Monitor's;s buffer (CCBUF) and moved to our buffer (CBUF).

When we finally have a control command image, we normally have to skip over the part that says "FORTRAN4" (or whatever). However, under UTS we must not do this, because there is no such thing; the options come on a separate line following our prompt (a one-character write of the character). When we ask for a control command image, it would be more consistent if all the Monitors give it to us starting at the point immediately following the introductory word (if any).

First we intialize all the option triggers off. Then we scan the control image. If no options are specified, and we are on-line, assume PS and GO. In addition, regardless of what options are specified, if we are on-line, always assume GO. Options are scanned up to a non-alphanumeric character. The options that include non-alphanumeric characters (BC,SB) are handled by a special section that scans their arguments. If any option is not recognized (or if a special option has improper arguments), a vertical bar (called in the listing a vertical slash) is inserted into the slash buffer at the character position corresponding to the illegal character. Several such errors may result in several slashes in the slash buffer,. Then, after all the options have been scanned, if there were any errors, we print the control image, the slashes, and an error message on the LO device. If LO and DO are different, we also print those same lines on the DO device. Then we blank the slash buffer. (This is post-initialization and is probably not a good idea. This blanking ought to be done before each line, but this is faster, and in a non-restartable mode of operation (see Restart, 4.5.2), it works properly).

4.3.4 Load overlay segment (PASS1ML, PASS2ML, PASS3ML)

These are called by the overall control code in the assembly FORTRAN4, to load in the three overlay segments, Pass 1, Pass 2, and Pass 3. These segments are called 1,2, and 3 under RBM. They are called PASS1F, PASS2F, and PASS3F under BPM/UTS. These names come from the external DEF names on the assemblies that begin each of those overlays.

On BPM/UTS, we can just do an M:SEGLD referencing the proper TEXTC containing the name of the overlay. On RBM, we have to build the overlay number into a CAL1,8.

Since these routines are called from POP language, exit is made via a BIR,IL INTERP.

4.3.5 Exit to Monitor (BPMEXIT)

This is used for normal (i.e., non-abort) exit from the compiler. It is called from the assembly FORTRAN4 when FOPTIONS encounters a new control card, or if SI is not equal to C and BC is not specified or the BC count is exhausted. It is also called by SIREAD when EOFs or EODs are encountered (also depending on the setting of the BC option). It exits by invoking an M:EXIT Cal. First it has to close the necessary files (see Close Files, 4.4.2).

Note that we return with register zero destroyed (we use it for our "A" register), but the Monitor is not adversely affected. We used to save the value of register zero on entrance and restore it upon exit, until we found we did not have to.

4.3.6 Abort to Monitor ABORTML, BPMABORT, ILPRABRT

All actual aborts to the Monitor are performed by ABORTML (which also has the alternate name BPMABORT). ABORTML is called by a number of places, both in and out of MERRIMAC. It can be entered with BRL or BAL or B; it does not matter, since it does not come back. ABORTML closes the same files as BPMEXIT (see Close Files, 4.4.2) and then aborts with an M:XXX Cal. This differs from an M:ERR Cal in that succeeding control cards in the same job will not (for the most part) be acknowledged. If a FORTRAN compilation has had such a serious error that it has aborted, it is not likely that the rest of the job is going to run right.

There is also a routine in MERRIMAC called ILPRABRT (Interpretive Language PRint and ABoRT), which prints an error message and then branches to ABORTML. Since this goes through the standard error message handler (PRERR1), the error messages should be of the standard format, i.e., with a first byte indicating severity, followed by the message enclosed in dollar signs. (Note: some of the XOS calls on ILPRABRT do not have the right format for their error messages, which might cause incorrect operation.)

4.3.7 Determine number of lines remaining on LO page NPRLINES

This is called only by the GNL (Guarantee N Lines) POP, which is used mainly in the Summary to assure that we do not print a heading on one page and its contents on the next. Most tables begin with a GNL 3. The little table at the very end that shows program sizes begins with a GNL 9. Also, in Pass 1, when we print a source line with an error, we use a GNL 3 to get the source line, the marker, and the error message on the same page.

RBM does not provide this information. On BPM/UTS we get it with an M:DEVICE M:LO,(NLINES). If the value comes back zero, it does not really mean that no lines are left. It means that the value does not apply, for one reason or another, so we substitute 1000, so that GNL will not think it ought to do a page eject. This is also what we do under those Monitors that do not tell us how many lines are left.

4.3.8 Test sense switch SENSWICH _ Used only by F4DEB

This routine is supposed to test the hardware sense switches and allow the debug package to terminate long-winded operations when the user signals with a sense switch. This only makes sense on a dedicated, free-standing system, which is how the compiler was checked out, so free-standing MERRIMAC does just that. If the debug package is fixed to work in an on-line environment, it will have to do something with some of the special "escape" characters, since there are no sense switches on a time-sharing terminal. Once upon a time, we used the Monitor's simulated sense switches, but this decreases their usefulness, because we need the sense switches to be an immediate signal to stop, whereas to change the setting of the s;Monitor's;s sense switches, you have to key in a message to that effect, and meanwhile the compiler is running off in some strange direction and you cannot stop it.

What this routine currently does is get into Master mode so that it can actually test the hardware sense switches. However, the use of debug under BPM as currently implemented is so messy that it will never be officially acknowledged, so this routine is essentially nonexistent. It is supposed to receive, in register 3, the number of the sense switch to be tested, and return with the condition codes such that a BANZ branches if the switch was set.

4.4 SPECIAL OPERATIONS

4.4.1 Open files INITIAL)

INITIAL is actually the initialization routine for MERRIMAC, but the main thing it does is open files. The only other things it does are: (a) initialize the DCBREQ count to zero (see LOWRITE, Section 4.2.2); and (b) set up the traps (see Set Up Traps, 4.4.3).

In theory, the compiler should not have to open any files. This is handled automatically by the Monitor the first time you try to use a file. Because of various requirements, however, we find it necessary to open (and in some cases play around with) most of the files we are going to use.

First, if it is RBM, we have to open the GO file and skip forward to avoid replacing any GO data that has already been created either by FORTRAN or by other processors. (This is handled automatically by BPM, as we think it should be.) Then we have to set up the scratch files, but before we do, I will digress into a history of the names of the FORTRAN IV scratch files.

Historical Note: There is much confusion in both the manuals and the listings as to what the names of the FORTRAN IV scratch files are. Both M:X1/M:X2 and F:X1/F:X2 appear, though at no time were both in existence. I believe the correct chronology of events is as follows: Originally we called these files M:X1 and M:X2 and we REFed them, assuming they would be provided by the Monitor. They were, but there were problems with them. Under BPM, for example, the DCBs were each 90 words long, which was more than we needed. In addition, these files were defaulted to, for us, inappropriate devices in inappropriate modes. We could have actively opened them and made them the way we wanted them, but then we would be defeating any user who had ASSIGNed them the way <u>he</u> wanted them. The only way to get around that would have been to go into the Monitor and examine the ASSIGN Merge table and come up with something consistent with what a user might have requested, and this was just too much hassle. On top of that, different Monitors and environments were being created that defaulted M:X1 (and M:X2) in various other ways. So what we did was change the names to F:X1 and F:X2 and define the DCBs ourselves within MERRIMAC. Thus they had the default conditions we wanted, but the user could reassign them if he so desired.

Now enter RBM, stage left. The DCBs that we had defined for F:X1 and F:X2 were too long and full of non-RBM information for RBM to handle (though they were as short as we could get by with under BPM). We could not use those DCBs for RBM. So rather than define <u>two</u> sets of scratch file DCBs, we decided to go ahead and use the default scratch file assignments in RBM, which were handled appropriately for us. However, they were called M:X1 and M:X2. So we went back to calling our scratch files M:X1 and M:X2. But we still could not use BPM defaults for these files; we had to define our own. We could not define them in MERRIMAC, because then they would override RBM's, so we defined them in a separate assembly, called X1X2DCBS. This assembly is not included in the system under RBM, but should be under BPM. In the process of making all these changes, not all the "F:" names in the listing have been caught and changed properly. So F:X1 and F:X2 have been EQUed to M:X1 and M:X2, so that either set of names will work in MERRIMAC.

Now enter UTS, stage right. In an on-line environment (or in Dual Batch Mode), there cannot be just one scratch file defined if there is more than one user. Since we are not using the Monitor's default scratch files, it does not know that our scratch files should be considered unique for each user (whereas it knows this about, say, the GO file or the LO file). Multiple users would all be writing into the same files, with disastrous results. Thus we have to

make the scratch file names unique by tacking the two-character User's ID (from the JIT) onto the characters "SX1" and "SX2". End of historical note.

On RBM, then, we do a couple of M:DEVICE Cals to make sure that the default scratch files are Packed Binary and are 202 words long (which is the size of our scratch records).

On BPM/UTS, we open the scratch files with the names modified by tacking on the two-character User's ID (as described above).

In either case, we set up the RBMTRIG to indicate whether or not we are running under RBM, and if we are <u>not</u>, then CJOB points to CCJOB, the JIT pointer. Under RBM, CJOB is zero.

Next we have to open the M:SI, M:LO, and M:DO DCBs. Normally we would not have to do this, because it will happen the first time we use them. However, on these particular DCBs, before we use them we have to do some correspondence tests to see if they are set in strange ways (such as equal to each other, or to typewriters, or to NO), and the Monitor does not provide correspondence tests that know that if the DCB is not opened yet, if it ought to be, or if the test will not be meaningful. (See Test For Correspondance Of Files, 4.4.4.)

Next we print the compiler version on the LO device and set up the traps (see Set Up Traps, 4.4.3).

Finally, we have to do one of the tests mentioned above that caused us to have to open some of our files. In this case, we have to make sure that SI is not assigned to NO. (We think that this should be taken care of by the Monitor.) However, you cannot just check to see if SI is assigned to NO. First you have to make sure that SI is not assigned to a file; otherwise the process that tests for "NO" is not meaningful. Then we have to check whether SI has an op label or device type assignment before we can accurately determine whether the assignment is to NO. If it is, we output a diagnostic.

4.4.2 Close files (BPMEXIT,ABORTML)

When leaving the compiler, either on an abort or on normal completion, we close and save the BO, GO, LO, SO, and DO files so that appropriate things will be done with them afterwards by the Monitor. (For example, if we do not specify SAVE, then the file will not be saved even if the user specifies SAVE on his ASSIGN card.) This could probably be centrally handled by the Monitor also, since what processors want to happen to these commonly used files is probably pretty standard. Note that we do not close or save the scratch files, since we do not care what happens to them after we are through.

We do not close files between consecutive compilations: only when exiting to the Monitor.

4.4.3 Set up traps (INITIAL)

We want to ignore fixed point overflow, but abort on any other abnormal hardware conditions. This does not include floating point overflow; the compiler uses no floating point instructions (see Instruction Sets, Section 4.5.7), so does not have to provide a trap handler for floating point traps.

Historical note: BPM used to provide default traps the way we wanted them, ignoring fixed point overflow. It was later changed to trap on fixed point overflow, so we added this M:TRAP Cal. If a survey were taken on this, we would vote to put it back to the old default.

4.4.4 Test for correspondance of files

These routines are used to test whether certain files are assigned to certain devices or are assigned to the same thing as another file. Some of this is truly useful to the compiler (so it can proceed differently if it is on-line, for example) and some is needed only because certain Monitor features are not very device-independent and we have to do some of the input/output processing on certain devices (notably typewriters). Of the following six routines, only LOEQFILE and TYTEST actually do any work. The other four call one of these two.

4.4.1 Test if SI equals LO (SIEQLOML)

This routine is used by Pass 1 to determine whether to print source lines on the LO device. If LO equals SI, they have already been printed (typed) by the user. If we are on-line, we call LOEQFILE to make the test; otherwise we assume that SI does not equal LO.

4.4.4.2 Test if LO equals specified file (LOEQFILE)

This routine tests whether a specified file is assigned to the same device as LO. It returns with the condition codes set so that a BE instruction will branch if they were assigned to the same device. There is an M:DEVICE Cal in BPM to test for the correspondence of files, but there is not in RBM, so the work must be done by hand. In either case, the test does not always give the answer we want. Before we can ask the Monitor whether two DCBs are assigned to the same device, we have to make four checks to be sure that it is safe to ask. If either LO or the specified file is not assigned to a device, the test will not work, so we check for that and exit FALSE. If LO and the specified file are both the same Op Label or the same device code, then we exit TRUE This sounds like we have done the whole test already, but there is still the case where one is assigned directly to a device and the other is assigned indirectly through an op label. This does not apply on-line under UTS, so we test for that and exit FALSE.

Assuming that there is still testing to be done, it has to be done by actually going down into the DCT index table. Under BPM, you are not allowed to do this yourself; you must use an M:DEVICE Cal to do it for you. Under RBM, there is no such thing as an M:DEVICE Cal, so you must do it yourself.

4.4.4.3 Test if LO equals typewriter LOEQTYML

This routine is used only by SUMMARY to decide how many columns of output to use in its summary tables. A console typewriter will not carry as much width. LOEQTYML is called from POP language with a QBL (or BRL) and returns with answer TRUE or FALSE. It uses TYTEST to do the work, supplying the argument M:LO.

4.4.4.4 Test for SI from typewriter TYINPUT,SIONLINE

These are used only in Pass 1, and mainly to assure that statements that are typed in from an on-line console and are found to have an error are always deleted so that the user can replace them. TYINPUT is called from POP language with a QBL and immediately calls SIONLINE. SIONLINE is called in machine language. It first tests to see if we are on-line, and if so, calls TYTEST to do the work, supplying an argument of M:SI.

4.4.4.5 Test if specified file is assigned to a typewriter TYTEST

This routine is used in a variety of places to see if input (or output) is coming from (going to) a typewriter. There is no direct way to inquire whether this is the case. In fact, five different tests have to be performed. First, if the specified file is not a device type (i.e., it is a file), then it is obviously not a typewriter. Otherwise, if it is assigned through an op label, check if that op label is OC. If it is not an op label,; check for both of the device codes for "typewriter" and "user console".

This routine skips if FALSE, and executes a normal return if TRUE.

4.4.5 Enter Master mode MASTER _ Used only by F4DEB)

The debug package has to get into Master mode to do privileged operations, such as storing into protected code (which it does when patching) or testing the hardware sense switches. It did not have to do that, of course, in a free-standing environment, and probably, if debug is ever made to run under UTS, the whole compiler will be unprotected, so that the debug package can store at will.

4.4.6 Enter Slave mode SLAVE _ Used only by F4DEB)

The debug package has to get back into Slave mode any time that it has gotten into Master mode.

4.5 Other Considerations

4.5.1 Entrance from Monitor

The compiler begins execution in machine language at the first instruction of the assembly FORTRAN4. in Pass. This is determined because FORTRAN4 is the only assembly with a transfer address on its END card. In other words, it is the "main program" of the compiler.

It is not significant (for loading purposes) that this assembly is called FORTRAN4. Nor is it necessary to load this assembly first; in fact, PASSOF is loaded first.

4.5.2 Restart

In theory, the whole compiler is designed to be self-initializing, so that it can be restarted at any time by branching to FORTRAN4. This was verified during checkout when the compiler was operated free-standing, because it was common practice to restart in this manner. There are apparently some areas in MERRIMAC that do post-initialization. However, this MERRIMAC was designed for operation under various Monitors, none of which provide for restarting processors. If the compiler were to be run under such a Monitor (e.g., a small, stand-alone type of system), MERRIMAC would probably have to be changed anyway, and this is one of the areas in which it would have to be changed.

4.5.3 Consecutive Compilations

There are two aspects of this that need to be considered:

A. When source programs and control cards are coming from the same device (i.e., SI=C), the user ought to be able to place several source decks behind a single FORTRAN control card, and have all the decks compiled.

B. When SI is different from C, the user ought to be able to either compile several source decks at once or compile them one at a time. For example, suppose that a tape (or file) contains four source decks, with no EOFs or anything separating them. The user may want to compile all four of them with one control card. Or he may want to compile only one or two of them. Or he may want to change options between the programs (like get an object listing only on the third one). The tape might be an unlabeled tape (prepared elsewhere) that is terminated with a single EOF. Or with a double EOF. Or nothing at all (just the four programs and then garbage). Or the four decks may be separated by single EOFs with a double EOF at the end. Or with nothing at the end.

This compiler will handle all of the above situations, and I believe it is the only Xerox FORTRAN compiler on any machine that will. The other FORTRANs (even on Sigma 7) approach consecutive compilations from a variety of different ways. It may be instructive to look at the history of this subject.

1. The 9300 system was the most straightforward system, though it was not quite as powerful for the user because there was no BC option. The compiler was very simple. It always compiled one program and then returned to the Monitor. This meant that you had to have a separate control card for each source deck that was coming from another input file (i.e., SI not equal to C). However, it was still possible to compile several decks behind a single control card (when SI=C), because the Monitor took care of this. When a processor returned, the Monitor read a card from the Control device. If it was a control card, it proceeded normally. But if it was not a control card, then it assumed it was a source card and did two rather clever things: it transferred control back to the previous processor (which was still in memory) so that processing could continue and it saved the card it had read. Then, when the processor asked for the first Source Input card, the Monitor gave it that saved card instead.

   This was an excellent approach. The Monitor automatically handled the multiple-source-decks problem for all the processors. If the 9300 compiler had had a BC option, it would also have been optimum for the user.

2. The 9300 RAD Monitor did not have this capability. Rather than change the compiler (which did not have its Monitor interface centralized into one assembly), a special interface routine was used between the Monitor and the compiler. When the compiler exited, the interface routine simulated the Tape Monitor's actions of reading the Control device and saving the source card.

3. The 900 Series RAD Monitor also did not have this capability, but the 900 Series FORTRAN IV would not have used it anyway. That compiler did not stop after a single program; it kept on reading from SI until it reached an EOF. This simplified the situation of multiple decks behind a single control card (where SI=C), but it made it impossible to compile individual programs from a tape or RAD file, or change options between programs, unless they were separated by EOFs. This philosophy (in one form or another) has been followed by all the other FORTRANs produced since, except for the Extended FORTRAN IV.

4. The Sigma Monitors also do not automatically handle multiple source decks. The same procedures are simulated, however, in MERRIMAC. Following an END card, FOPTIONS reads from the C device, returns to the Monitor if a control card is found, or saves the card if it is a source card, and sets the SICTRIG. This tells SIREAD to get its first card from a special buffer. (This is all described in the two sections, "Interpret control card options", 4.3.3, and "Read a source line from SI", 4.2.1.)

The BC option then takes care of the SI not equal to C cases. Without BC, each control card compiles one program, thus giving but also requiring complete control. Plain BC allows you to compile an unknown number of programs and terminate on a single EOF. BC(n) allows you to compile a specific number of programs, and it ingores single EOFs. BC(n) with an enormous value of n allows you to compile an unknown number of programs (with or without single EOFs between) and terminate on a double EOF.

5. For comparative purposes, I believe both FORTRAN IV-H and FLAG always read multiple programs from SI, but FORTRAN IV-H ignores single EOFs and terminates on double EOF, while FLAG (with no BC) terminates on a single EOF. With BC, FLAG does the same as FORTRAN IV-H, but not the same as what FORTRAN IV does with BC. FLAG has no BC(n).

There are a couple of other considerations related to consecutive compilations. One of these is that, in order to really stop after each program and go read from C, the compiler must not read past an END card. Unfortunately, the FORTRAN language is defined such that any statement may have a continuation card, and you do not know whether there is one until you go look at it. What do you do about the following situation?

```
      END

     X     FILE 3
```

What we do is prohibit it. However, the continuation mechanism, which is built into the low-level input scanning routines, has to be temporarily turned off to prohibit them from looking for continuation cards on END statements. This is described in "Next Active Character" (Section 2.2.2) and "END Statement" (3.4.2).

Another curious problem arises. Suppose the user is reading 026 keypunch source decks, from the same device as the control cards (i.e. SI+C). He specifies the FBCD option so that the characters will be converted to EBCDIC. However, this applies only to the SI device, and if he has multiple source decks following a single control card, the first card of each program will be read via the C device, of which FBCD does not apply. It will therefore be in the wrong character set and will probably get syntax errors. There is no clean way around this (since the compiler does not know that FBCD is in effect; if the Monitor were handling this process, it could do the conversion in the buffer, once it found out that the card was an SI card rather than a C card). The user has to get around this either by using the BC option (which will keep the compiler reading from SI) or by preceding each program with a comment card, on which the character conversions will have no effect.

4.5.4 Paging conventions

A fair amount of design has been done on when to do page ejects, in order to produce a clean listing and still conserve paper. A compilation consists

(potentially) of three parts: the source listing, the object listing, and the summary listing. If there is a source listing (i.e., if LS was requested), we page eject so that it begins at the top of a page. If there is an object listing, we also page eject before that, so that it begins at the top of a page. Similarly, if there is a full summary (i.e. neither PS nor NS has been requested), we page eject to start it on a new page. If only a Partial Summary (PS option) has been requested, it is not preceded by a page eject. A partial summary is so small that it is not worth wasting a whole page on. Instead, it is printed just below the source or object listing (if any), with three blank lines of separation.

One of the capabilities we wanted to preserve was that of running a series of compilations, each with little or no listing, without doing a lot of unnecessary page ejects and thus producing a lot of nearly empty pages. For example, a typical way to recompile a file of programs is with the control card

```
!FORTRAN PS,BC
```

PS is a useful way to do this, because it gives the user some information about each program, but without page after page of listings (all of which he saw the last time he compiled). With PS he gets the first and last line of each program (as well as each SUBROUTINE, FUNCTION, or ENTRY statement) followed by the sizes of the various parts of the program. If there are any error diagnostics, he will get these too. Although he would also get the diagnostics if he ran with NS (No Summary), it would be hard to correlate them with the proper program.

In either case, as long as no source, object, or full summary is printed, the compiler will never page eject, so the programs will follow one another on the same page. This seems more appropriate than using a whole page for each. It also allows programs to be compiled with NS and have nothing printed (assuming there are no errors).

There are more extensive things one could do to produce nice listings if one were setting out to design a monitor/processor system as a whole. In fact, the problem of when and where to page eject can become quite complicated. The remaining paragraphs in this section are a record of the speculation that was done on this subject during the design of the compiler.

The first consideration is where the !FORTRAN control card gets printed. Ideally it ought to appear on the same page and just ahead of the source listing (or whatever) to show what the options were. It can be useful to know, for example, that the ADP option was specified, or that the S option was not. As it is, the !FORTRAN card is on the previous page, which probably gets torn off. If it happens that the !FORTRAN card was printed at the top of a fresh page, our page eject will result in a blank page. But there is no way we can know whether the !FORTRAN card was printed at the top of the page, and if we did not give a page eject, we might start our listing near the bottom of a page filled with other, unrelated information, which would not be very attractive. Ideally, the Monitor would do the page eject for us and print the

!FORTRAN card at the top of the page, but the situation is not quite that simple, as we shall see.

At the end of a compilation, no page eject is given. We may be about to do another compilation, and we do not want one page eject after this one, and another before the next one (thus producing a blank page). Clearly when the compiler is handling the consecutive compilations, it could keep track of this. However, if there are several !FORTRAN cards, we have to return to the Monitor at each one. The Monitor then loads the compiler afresh, so that it is impossible to remember whether an initial page eject should be given. To avoid the blank page, therefore, we give no page eject after compilation. This means that subsequent non-FORTRAN operations, such as loading, will begin on the same page as the summary (or whatever the last listing was). The same is true of the !FORTRAN card for the next program; it appears at the end of the previous program. Ideally, we should give a page eject at the end of compilation if we have printed anything substantial, and the Monitor could print the !FORTRAN card at the top of the next page and then leave the page there, so that we could start right after it. This requires the Monitor to know that it is at the top of a page (otherwise it would have to do a page eject, as indicated previously).

Now consider the case where no real listings are being printed. As described above, we want to avoid producing a series of virtually empty pages, so we issue no page eject (we never page eject after compilation, and we only page eject before compilation if a source, object, or summary listing has been requested). However, ideally the Monitor would get to top of page when it calls the compiler in. If it did that in the 'no listings' case, and there were dozens of !FORTRAN cards, that would produce the unwanted dozens of empty pages. The really ideal situation could be approached in the following way:

A. When the compiler is first called in, the Monitor goes to top of page and prints the !FORTRAN card.

B. The compiler does not page eject before its first listing.

C. The compiler does page eject after each program, but only if one of the three full listings were produced. If no significant listing was produced, then we are willing to start another compilation on the same page, but only if it, too, has no significant listings.

D. When the compiler starts a subsequent compilation, it examines the options to see if any significant listings will be produced. If not, it stays where it is. If so, it goes to top of page, unlessit is already there.

E. If the subsequent compilation has its own control card, then the Monitor must recognize that this is a "subsequent" control card, and not go to top of page, but leave that for the compiler to decide.

Even the above is not quite perfect . When a significant listing follows an insignificant listing, the second !FORTRAN card will appear at the bottom of the first listing, rather than at the top of the second. I see no way around

this, other than to make the compiler print the control card. Maybe it ought to do so anyway.

4.5.5 Error severity

The compiler uses a different scale of error severity internally than that which appears on summary listing and interfaces with the rest of the system. The internal scale is a hold-over from the 9300 FORTRAN compiler. The two correlate as follows:

Internal	External	Meaning
0	0	No error.
1	4	Minor error, from which the compiler can recover in a manner that is likely to be what the user had in mind.
2	7	Major error, from which the compiler can recover by making some sort of assumption, though it is not likely that the program will work correctly.
3	10	Major error, and the statement has been deleted.
4	15	Abort error, making it impossible to continue compilation. These are rare and are caused mainly by hardware failures.

The external severity is consistent in interpretation with that used in the run-time. The severity is output as the last byte of each program's load module, so that the loader can decide whether or not to load a program, based on the highest severity encountered in any of its parts. The SL (Severity Level) option on the ±LOAD card allows the user to specify the severity he will tolerate. The default value is 4, which means that any major error will prevent loading (unless overridden by the SL option); programs with only minor errors will proceed to execution.

4.5.6 64K restriction

The compiler's interpretive POP language is implemented using the Sigma INT (Interpret) instruction. This instruction operates on only a half-word address (i.e., 16 bits instead of 17). That is, the INT instruction itself uses a 17-bit address, but the word being interpreted (i.e., the POP "instruction") has only a 16-bit address field. (The first 4 bits go into the condition codes, the next 12 are the POP location, and the last 16 are the operand address. See Section 1.6). This means that no POP can directly

address any location above 64K. To assure this, the compiler checks to make sure that its highest location is below that point. If not, it prints an error message and aborts. Note that the working storage for lists can be anywhere, because data in lists is always addressed via the list parameters, which are down within LISTDEFS. The POPs either address indirectly through the BOTTOM of a list or, more commonly, the POP specifies a list number and the interpreter finds the list and the appropriate item.

We feel that it might be a good idea for a processor (or any load module) to be able to specify to the loader that it cannot be loaded above 64K. There are other processors (like Meta-Symbol) that have this problem, and this would enable the operating system on huge systems to decide who to put where.

4.5.7 Instruction sets and assembly language

The entire compiler is written in the Sigma 5 instruction subset, with no floating, decimal, or privileged instructions (i.e., all the assemblies use SYSTEM SIG5). This turns out to have almost no adverse effect on the compiler. It does not need privileged instructions, because the Monitor handles all privileged operations. It does not need decimal or byte string instructions, because the FORTRAN language very seldom needs either kind of manipulation. And it does not need the floating point instructions because it actually does very little floating point arithmetic. Most of that is done by the object program. The compiler itself mainly needs to convert values between binary and decimal, for which the floating point instructions are not necessary.

The object code generated by the compiler is also in the Sigma 5 subset, but does make use of the floating point instructions (the object listing shows SYSTEM SIG5F). The same considerations apply concerning decimal, privileged, and non-Sigma 5 instructions. Note, however, that in-line assembly lines (S in column 1) may contain any of these instructions, and they will be correctly generated. It is the user's responsibility to know what he is doing with these. (If the S option is specified, the object listing shows SYSTEM SIG7FDP instead of SYSTEM SIG5F.)

One of the reasons that we stayed in the SIG5 subset was to avoid possible interface problems with various Monitors. For example, on a 9300 without floating point, the Monitor provided a simulation package at run-time, but not at compile-time, an unobvious fact that caused some difficulties.

The compiler assembles in Meta-Symbol. It will not assemble in Macro-Symbol, due, among other things, to the extensive use of lists in Procs. The generated object code that is shown on the object listings is Meta-Symbol compatible, except for a couple of areas where the compiler has to choose between being FORTRAN-oriented and Meta-Symbol-oriented. These are:

A. Most constants are listed in their FORTRAN form rather than in Meta-Symbol form. For integers and most hexadecimal constants, the form is the same. All of the others are different, as shown in the Operations Manual.

B. Symbols can be defined at more than one place in a FORTRAN object listing (for example P, C, and E in the register field are not the same as user variables with the same name).

Macro-Symbol has one additional incompatibility, in that it requires REFs and DEFs to precede their first use, whereas the compiler lists them last.

4.5.8 Control sections and protection type

Most of the compiler itself is CSECT 1 (write-protected control section). Two principal assemblies, FORTTEMP and LISTDEFS, are CSECT 0 (unprotected). The assembly X1X2DCBS, used under certain Monitors, is DSECT 1 (write-protected dummy section).

The generated object code can contain three kinds of control sections: CSECT 1 for the instructions, CSECT 0 for the local variables and temps, and DSECT 0 for COMMON and GLOBAL (which is treated like COMMON). Normally, every program contains both a CSECT 1 part and a CSECT 0 part. It may or may not contain one or several DSECT 0 parts, depending on whether COMMON has been used.

There have been several problems in getting the loaders to accept FORTRAN generated programs as part of the run-time library (i.e., the user wants to write a library routine in FORTRAN). Section 4.5.9, below, also discusses these. One of the problems is that a library routine is not allowed to have more than one protection type. It is impossible to compile a FORTRAN subprogram that has only CSECT 1 protection type. Every standard receiving sequence requires at least one temp (for the return address). Therefore, a normal FORTRAN compiled subprogram cannot be included in a library. To get around this, the user must compile with the NMP (No Memory Protection) option, so that all of his program will be CSECT 0. This means, of course, that his library routine will consist of unprotected code, and furthermore that it cannot be used in a time-sharing "public" library. The best way to get around this is to compile the program and then copy the generated object code into a Meta-Symbol program and assemble it, moving the data (unprotected) storage into a separate assembly.

The NMP option is also required when the user wants to read into a Hollerith field in a FORMAT statement, since FORMATs are normally contained in the protected CSECT 1 area.

4.5.9 Program size, REFs and DEFs

The names and sizes of all the control sections in the object program are output at the beginning of the binary deck. Thus the loader is able to know the size of each section before it begins loading. (On the 9300, the assembler did not output program size at the head of the program. This forced the loader to waste memory loading the program in one place and then sliding it around after it found out how big it was.)

With one exception, the REFs and DEFs are output immediately following the above control section declarations (i.e., ahead of the main body of code). Note, however, that the object listing shows the REFs and DEFs (as well as some of the control sections) at the end of the program, even though they are actually output at the beginning. This is to improve the readability of the listing. These items are simply not as important as the generated object code, and we did not want to clutter up the front of the object listing with them.

The one exception to the output of the REFs and DEFs is caused by a restriction in some of the loaders. Although the standard binary language allows the REFs and DEFs to come where they are (or indeed almost anywhere), certain one-pass loaders, when searching the library, discontinue looking for DEFs after encountering any control section declaration. Thus there was another problem in writing library routines in FORTRAN Section (4.5.8): The routines would not be recognized, because their DEFs would not be processed. Therefore, it was necessary to move the primary DEF (i.e., the name of the first subprogram, if there are internal subprograms) ahead of the control section declartions, thus becoming the first thing output. This is only done on FUNCTIONs and SUBROUTINEs, not on main programs or BLOCK DATA subprograms (which cannot be used in the library).

Unfortunately, it is not quite that simple. A DEF has two parts: the name being defined, and its value (i.e., the location at which it is defined). This value is, of course, relative to the beginning of the program control section, which has not been declared yet. This is also not permissible. (These loaders cannot process forward references.) Thus, although we normally output a DEF as a two-part item, containing name and value, here we have a situation where the name must precede the control section declarations and the value must not. Some finagling is necessary, therefore, to get the name out by itself and retain the value for outputting later. See also Start Binary Output, 2.28.1, on this subject.

One final item concerning DEFs: the FORTRAN language provides for "names" for programs and COMMON blocks, but it does not provide for a naming the local storage area. Typically, the variables and temps have always been part of the program and could be identified at run-time (e.g., for debugging) relative to the start of the program. In the Sigma environment, however, protected code and unprotected data cannot lie in the same control section, and unless each control section has a name, the data within it cannot be referenced by name at run-time. Therefore, a DEF was added at the origin of the local storage control section. It begins with "V#" followed by the primary name of the subprogram. In the case of main programs, it is "V#4MAIN". Note that if the subprogram name is longer than six characters, the variable section name will be longer than eight characters. This is the only place in the FORTRAN system (internally or externally, compiler or library) where a name of more than eight characters appears.

The V# name is shown in the object listing on a DEF line, but it is not shown in the label field of the CSECT defining the variable control section. This should be added.

4.5.10 Run-time I/O unit numbers and DCBs

The FORTRAN language provides that I/O unit numbers are integer values. Typically, the compiler generates code to transmit the value to the library I/O routines. For example, consider the following object code:

```
WRITE(3)  X

          LI,9      3

          BAL,9     9BINWRIT

          BAL,6     9IODATA

          SNGL,E    X
```

On the other hand, the Monitor deals with unit numbers not as numbers but as names (character strings). Thus the library I/O routines need to convert the integer value into EBCDIC, preceded by "F:". Thus, for example, the unit number 3 is referred to at run-time as F:3, and it is attached to a DCB. Ideally, this should not be necessary. If the Monitor refered to DCBs by number, as well as by name, the library could just pass the number that it receives on to the Monitor. But this pertains to the library, rather than to the compiler, and it is pretty straightforward compared to the problems that the compiler has.

Most unit numbers have to be ASSIGNed by the user. The act of ASSIGNing them causes them to be created, so that the Monitor will know about them when the library wants to use them. There are also, however, a set of "standard" unit numbers, which the user does not have to ASSIGN. They are supposed to be "built into" the system. For example, unit number 108 (or DCB F:108) is automatically assigned to the LO (Listing Output) operational label, unless the user chooses to ASSIGN it otherwise. The problem is that these standard unit numbers are not built into the Monitor, but into the loader, which means that the loader has to know which ones are going to be used so that it can create them. The library cannot just hand the Monitor a reference to F:108 and expect that it knows what to do with it. The necessary DCBs have to be REFed by the compiler in the object code, to instruct the loader to create them. Therefore, whenever one of the standard unit numbers is used, the compiler remembers this (on the REFLIST) and generates a REF to the corresponding DCB. See 2.23.

Suppose, however, that the program uses a non-constant unit number (e.g., WRITE(N,5) X). The compiler does not know what value N is going to take on at run-time, but if it takes on one of the standard values, the compiler was supposed to have REFed it. This is one of the problems in not having the standard units truly built into the Monitor. One way to get around this would be to REF all the standard units when this happens, but it is unlikely that they will all be used, so the others will be wasted space. We can do slightly better than this. The WRITE statement REFs only the standard output DCBs, and the READ statement REFs only the standard input DCBs. The information

concerning which DCBs are for input and which for output is kept on the UNITLIST (a static list) in P2P3ROOT. This still means some wasted DCBs will be REFed, but not as many.

There is still another problem to be dealt with. Some of the library routines need to do I/O operations that the user does not specify unit numbers for. For example, the 9STOP (which processes the STOP statement) wants to print on LO (so it used F:108); 9PAUSE wants to type on the operator's console (M:0C); 7ERROR wants to print its run-time error messages on the diagnostic output device (M:DO). These library routines (and others that do the same sort of thing) may or may not be loaded in any given program, so originally each of them REFed any DCBs that it needed. Thus if 9STOP were loaded (which it usually is), F:108 would be called in even if the user did not reference it. Subsequently, certain Monitors disallowed library routines REFing DCBs. This means that though these routines need certain DCBs, they are not allowed to REF them; the compiler has to REF them, even though there is no way it can know which ones will be needed. (For example, the program might call SINF, which calls 9SIN, which calls 9ERROR, which calls 7ERROR, which needs M:DO. How can the compiler know all this?) So the compiler unconditionally REFs the three DCBs, F:108, M:DO, and M:0C.

What is the maximum size of a unit number? The Monitor/Loader is restricted only by the length of the resultant EBCDIC character string, which apparently can be 31 under BPM and 11 under UTS. To be safe, the compiler and the FORTRAN documentation restrict the user to the range 0 through 9999. The actual limit would first be reached in the library routine 7UNITADR, which has to convert the number to a character string. It allows only a half-word, thus limiting the value to 65,535.

4.5.11 Direct run-time monitor calls

The compiler never generates direct Monitor calls in the object code. The Monitors change too often and there are new and different ones introduced, so rather than have to change the compiler when this happens, all of the actual Monitor interface at run-time is done by the library routines.

One of the places where it had been suggested that we might use direct calls was the use of the CAL instruction to pass I/O list items. Theoretically this could enable us to pass such arguments in one word, rather than using a call to 9IODATA followed by a word specifying the list item. (Actually, with the introduction of the C and E bits on I/O calling sequences, the average size of an IO call can approach one word anyway.) There were two principal reasons why we did not do this: first, it did not give us enough bits to pass the type and protectedness in addition to the address; and second, the item overhead for the Monitor to process the CALs was too high. It was both safer and faster to do it ourselves.

4.5.12 Hardware stacks

We also considered using the hardware stacks for our compiler lists (as Meta-Symbol does), rather than handling all the reassignment ourselves. There are several reasons why we chose not to do this.

A. The Monitor overhead to process the traps occurring on stack overflow is quite high. Meta-Symbol has been slowed down by its dependence on this service.

B. Stack size is limited to 32K. Although it seems unlikely that any ordinary list would ever exceed that (one would need 5462 symbols for the Symbol Table to do so), the Output List and Input List could do so more easily and we would have to check for this manually, since the hardware does not (it just proceeds with the wrong size).

C. The stacks are oriented towards being used only as push-down stacks, i.e., there is no real provision for taking data off the top (first in) end of a stack, and this is something that we do occasionally, such as in the TOT POP (Take Off Top).

D. The stacks provide only two parameters, essentially the Base and Bottom. (They actually provide a Limit too, but that is equivalent to the Base of the next list in our scheme.) There is no provision for the Start and Top, i.e. for creating files or (as discussed above) taking off the top. This can all be done manually, of course, (Meta-Symbol does it), as can most of the above problems, but by the the time you have done this much, you might as well have done it all.

E. A maximum of 12 words can be pushed onto a stack. Actually, this is not too much of a problem, because you can always use MSP (Modify Stack Pointer) to update the list size, and then move the words manually.

4.5.13 Arithemtic overflow and other traps

Since the compiler does not use any floating point instructions (see Section 4.5.7), no trap handler need be provided for floating overflow or underflow. Fixed point overflow is ignored; the hardware does the right thing in any cases that will occur in the compiler. Any other kind of trap (such as unimplemented instruction, decimal overflow, etc.) is an error and causes a Monitor abort. Thus the trap set-up call used in INITIAL (in MERRIMAC) is:

```
M:TRAP     (ABORT,ALL),(IGNORE,FX)
```

Note that originally, this was the default setting of the traps under BPM. It was later changed to abort on fixed point overflow, so the above M:TRAP call was added. If the Monitor should ever revert to the old default conditions, this M:TRAP could be removed, but it is probably safer to leave it in.

4.5.14 COMMON storage

Each block of COMMON storage is output as a DSECT (dummy section) with protection type 0 (unprotected). Note that the term "dummy section" derives from Monitor terminology and bears no relation to the term "dummy" as used in FORTRAN, meaning a formal parameter in a subprogram. A DSECT differs from a CSECT (control section) in that the same DSECT may be declared in several programs, and they all refer to the same thing. Obviously if several programs define the COMMON block ALPHA, they are all referring to the same alpha; that is the whole purpose of COMMON. This is implemented by having DSECT be neither a REF nor a DEF, but a little of both. A REF says, "I don't know where this symbol is; the loader will have to find it for me." A DEF says, "here is where this Symbol is, and no other program can try to define it." A DSECT declaration, on the other hand, says, "Here is the symbol for a block of storage that I need; use the same block for other programs requiring the same symbol." In other words, it is a DEF if the loader has not seen the symbol before, or a REF if the loader has. It also specifies a size. No program may specify a size larger than that declared in a previously loaded program; it may specify a smaller size.

There is a problem with the fact that the double-word instructions (used in the object code for processing Double Precision, Complex, and Double Complex data) can only reference even-numbered memory locations. When allowed to do so, the compiler carefully puts all double-word variables, constants, and temps in even locations. However, through use of COMMON and EQUIVALENCE, the user can request allocations that do not conform to this. All of these are now diagnosed and changed so that they do conform. Originally, however, the compiler was designed to recover from certain of these cases by taking advantage of a loader feature that was never implemented.

Consider the following three situations:

1	2	3
DOUBLE PRECISON D	DOUBLE PRECISION D	DOUBLE PRECISION D, D1
INTEGER J	INTEGER J	INTEGER J, J1
COMMON D, J	COMMON J, D	COMMON J, D, J1, D1

In the first case, the compiler can observe the rules if it assures that COMMON begins at an even location. In the second case, it can do so only by assuring that COMMON begins at an odd location. In the third case, beginning COMMON at an odd location takes care of D, but there is no way then to take care of D1. So the compiler must diagnose this and introduce a one-word gap between J1 and D1. See 3.8.2.

In order to handle case 2, however, the compiler has to be able to specify to the loader that it requires a DSECT to begin at an odd location, and this capability was never implemented in either the loader or the standard binary language. All DSECTS begin at an even location. Therefore, the compiler now

also diagnoses all cases that would require this. So what the compiler does in case 2 is to introduce a gap ahead of J, so that the block "essentially" starts in an odd location, even though it really starts in an even location. And the compiler produces a diagnostic in this event, however, because J is now in location 00001 of the block, while it is supposed to be in location 00000, and this might not interface properly with another program using the same COMMON area. See 3.8.7.

4.5.15 Blank COMMON

There are a couple of special things about blank COMMON. First, it is not declared as a DSECT with a blank name. It is given the special name "F4:COM". This cannot conflict with the name of any other COMMON block because of the colon in the name.

Second, the compiler goes to considerable trouble to prevent variables in blank COMMON from appearing in DATA statements. The simple cases of this are handled by the permissibility bits for DATA and BLANKCOM (see "Permissibility", Section 2.8.1). No variable that has appeared in blank COMMON can appear in a DATA statement, and vice versa. The more subtle cases arise through the use of EQUIVALENCE. For example,

```
DATA A /1.0/

COMMON B

EQUIVALENCE (A,X) , (B,Y) , (X,Y)
```

Not only does the compiler have to know that A cannot be EQUIVALENCEd to B; it has to carry this fact through chains of EQUIVALENCEs and thus, in the above example, disallow the EQUIVALENCE of X to Y.

The reason that all of this is done (aside from being a traditional restriction in other FORTRANs, which is not in itself sufficient cause) is to make F4:COM a special area that the loader can use for tables or whatever, and thus save space at load time. Especially on small machines, a program that is too big may be helped by moving variables into blank COMMON. The loader does not actually have to set aside and preserve the space for blank COMMON. It can use the F4:COM area for tables or even for the loader itself. The multipass loader (!LOAD) essentially does not even allocate any real memory for F4:COM. It just decides where F4:COM will be when the program being loaded is executed. What is in that area during loading is unimportant.

4.5.16 Special provisions for the DATA and NAME LIST statements

A DATA statement can specify that the same constant is to be loaded into a number of consecutive locations. For example,

```
DIMENSION AA(10,10)

DATA (AA(K,1), K=1,10) /1.0/
```

The binary language and the loaders provide a feature called "repeat load", whereby a load item can be specified with a repeat count, and the item will be loaded into consecutive locations, the number of times specified by the count. In the above example, the value of 1.0 could be specified with a repeat count of 10.

The compiler does not do the additional checking necessary to take advantage of this feature. It always outputs a new ORG and a new value item for each DATA pair (variable and constant). It does this even when the new ORG immediately follows the old ORG and is thus superfluous. (This is independent of the repeat load feature, since the value item may be different.)

In the 9300 binary language, there was an additional capability within the repeat load feature, that of specifying an increment. In other words, "Load this item N times, M words apart." This could be used, for example, if the array were intialized row-wise instead of column-wise:

```
DIMENSION AA(10,10)

DATA (AA(1,K), K=1,10) /1.0/
```

The 9300 compiler also did not take advantage of this feature.

There are no special provisions for the NAME LIST statement in the binary language or the loader, because the definition of the feature no longer requires them. On the 9300, all of the name lists in the main program and various subprograms were linked together into one combined name list, and it was the loader's job to link them together and tell the library routine 9INPUT where the chain began. For Sigma, the interpretation of NAME LIST was changed, such that each program has its own name list, independent of any other program's name list. This means that the compiler can handle the implementation entirely within the generated code for each program.

4.5.17 Top of page awareness

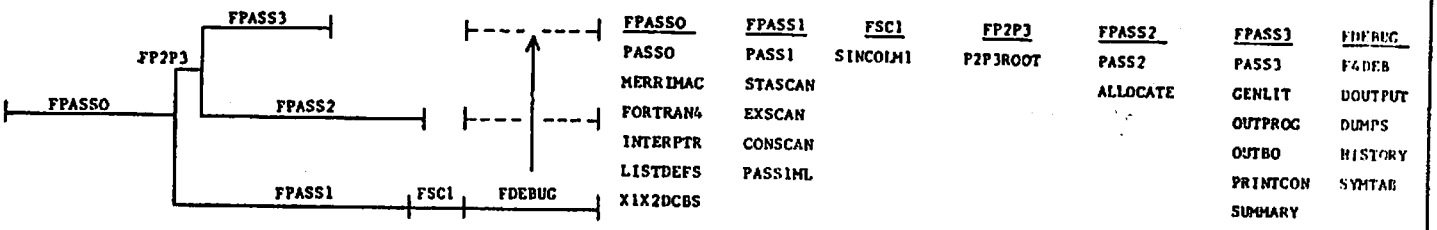
If the Monitor provided a test for whether you are currently at the top of page on the LO device, the compiler could do a few nice things, such as not printing blank lines at the top of a page. Actually, the compiler already avoids this in some situations. If it needs to do a PRL 4 (print the current line, followed by three blank lines), and there are only two lines left on the LO page, it will only do a PRL 2. It will thus avoid printing the two blank lines at the top of the next page. However, if the blank lines are not printed in conjunction with something on the previous page (i.e., if you do a PRL 1 on the last line, and then decide to print one or more blank lines), the compiler does not know it is at top of page, so it prints the blank lines.

It is possible that this could be circumvented by having the compiler constantly check how many lines there are left on the page, and set a trigger when it prints the last one, but this is not worth the effort. It might be, however, if there were a simple test for this condition.

4.5.18 Overlay Structure Details

The current overlay structure was shown in Section 1.2. It differs somewhat from the original structure that was designed for the compiler. In order to understand why things are currently where they are in the overlay structure, and in order to understand the proliferation of names that are used in describing assemblies and segments, it is instructive to look at the original structure.

In theory, the compiler's overlay structure should look like this:



As before, the diagram reflects approximately the relative sizes of the segments. This arrangement is simpler in some ways and more complicated in others. It needs some explanation.

P2P3ROOT is an assembly needed by both Pass 2 and Pass 3, but not by Pass 1. Therefore, it is in a sub-root for those two passes. (Actually, it is not entirely needed by Pass 2, as will be explained shortly.)

SINCOLM1 (which processes in-line assembly code) is designed so that it need not be called into memory if the S option is not specified on the control card. This frees more space for working storage in typical cases. This is one of the two reasons that in-line assembly language requires an option to be specified. The other is that such usage is highly machine-dependent, and we wanted to assure that programs that used it would be easily identifiable by looking at their control cards.

FDEBUG, the compiler debug package, is a little strange. Like SINCOLM1, it need be called into memory only if an option, in this case CDEB, is specified. Obviously if it comes in, SINCOLM1 must come in too. (At least, there is no way to save any memory by not bringing it in, because FDEBUG is going to have a constant origin; it cannot start at a different place depending on whether SINCOLM1 is in or not.) The strange part about FDEBUG is that if it does come in, it is supposed to stay in (along with FPASSO) while FP2P3, FPASS2, and FPASS3 are brought in between it and FPASSO.

Keep in mind that this is the way we would like the overlay structure to work. It has never actually worked that way, except on the 9300. A number of things have occurred to change the above structure. These are documented below:

1. There is no way to conditionally load FDEBUG, nor is it possible to load other overlays between it and FPASS0. Therefore, a debug compiler has to be a special system, with FDEBUG included in the root along with FPASS0. This is seldom of any real concern, since the compiler debug package is not currently operational under any Monitor.

2. Similarly, SINCOLM1 cannot be conditionally loaded, so it is now part of FPASS1. This means that the initial overlay is now the largest, which may or may not be important to any of the Monitors.

3. The names used to identify the assemblies PASS0, PASS1, PASS2, and PASS3, and the names used to identify their respective overay segments, have a long and colorful history of uncertainty. As explained in Section 1.2 ("Overlay Structure"), the informal names for the four overlay segments are Pass 0, Pass 1, Pass 2, and Pass 3. Within each of these passes, there is an assembly named after the pass. It is always arranged to be the first assembly in that pass. Thus the first assembly in Pass 0 is called PASS0, the first in Pass 1, PASS1, and so on. hese names are externally DEFed and are used by the compiler in cross-referencing itself. Originally these names were the first ones DEFed in the assembly, so that the assembly would be "known by" that name.

   To identify the overlay segments, we chose the names FPASS0, FPASS1, etc. We could not use Pass 0, Pass 1, etc., because on listigs the names would have to be capitalized and they would look too much like the names of assemblies, rather than the names of overlays. These names (FPASS0, FPASS1, etc.) still appear on the title cards of the assemblies (to identify which overlay they belong in) and also in the "dash eleven" writeups of the compiler, despite the fact that they really do not have (nor ever have had) any basis in fact. Neither system generation nor segment loading uses those names.

   The official names for the overlays derive from the names of the files in which the binary object decks (ROMs) of the various assemblies are kept. Each assembly's ROM is kept in a separate file, which is named by an ASSIGN card and has nothing to do with any DEFs in the ROM. Originally, the file names were exactly the same as the "true" names of the assemblies (i.e., the first DEF). The name of an overlay segment is automatically taken from the name of the first file within that segment. Thus, the name of the Pass 0 overlay would automatically be PASS0, the Pass 1 overlay would be PASS1, and so on. Unfortunately, at system generation time, it is not permissible for the name of any overlay segment to conflict with a similar name in any other processor or in any Monitor overlay that is called in by name. Thus there is a list of reserved names for overlay segments, and all four of the segments PASS0, PASS1, PASS2, and PASS3 conflicted with something.

   Therefore, the files containing these assemblies were renamed PASS0F, PASS1F, PASS2F, and PASS3F. This automatically renamed the overlay segments and removed the conflict. A better solution could be achieved if processors were allowed to name their overlay segments independently of

the names of the files within, or if the overlay segments within one processor did not interfere with those of another processor.

To summarize, the name PASS1 is the compiler's name for an assembly. (The other passes follow the same convention.) PASS1F is the official name for the assembly, because it is the name of the file in which that assembly is kept. PASS1F is thus also the official name for the overlay segment, and is used in system generation and by MERRIMAC when requesting a seg-load. FPASS1 is the compiler's unofficial name for the same overlay segment. Paragraph 9 below will describe yet another name (#PASS1F) associated with the assembly.

4. UTS is not able to accept a sub-root within any branch, so P2P3ROOT cannot be in a path that is common to Pass 2 and Pass 3 but not Pass 1. In order that the UTS overlay structure not have to be different from the BPM overlay structure, P2P3ROOT was unconditionally moved into Pass 0, thus increasing the size of the root.

5. Originally the compiler was not re-entrant. The temps were scattered throughout the various assemblies. When the compiler was made operational under UTS, the temps were all moved out of the individual assemblies into assembly FORTTEMP, which resides in Pass 0. A preferable situation would probably be to have a separate temp assembly for each of the overlays, so that local temps would not have to take up space in the root, but it is questionable whether this is allowed. This has been partially accomplished by re-origining the Pass 2 and Pass 3 temps to occupy the same space as the Pass 1 temps. As it turns out, however, the Pass 3 temps are by far the largest (especially since the binary output buffer was increased to 254 words to accomodate XOS binary format), thus taking up space in other passes that are already longer.

6. The scratch file DCBs, which were originally REFed and supplied by the Monitor, needed to be defined in a special assembly that is used under some Monitors and not under others. (This is explained in Open Files, Section 4.4.1.) This explains the assembly X1X2DCBS, which is also in Pass 0.

7. Because the debug package is not a standard part of the compiler, and because, in any case, reassembling the compiler to make a whole new system is a large job, it seemed desirable to provide a patch area, into which interim fixes could be placed, in between system reassemblies. There is a protected patch area, called #PATCH for instructions, and an unprotected (CSECT 0) patch area, called #PATCHT, for temporary storage by patches.

8. There is more to being able to patch than meets the eye. The patch area is in the root segment (Pass 0). Patches would like to refer to places in other segments (e.g., to branch back). Patches can only refer to symbols that appear in the REF/DEF table of their segment, and many of the assemblies in the other segments are not directly REFed by the root. To correct this, the assemlby "REF" is included in Pass 0. It REFs the names of all the assemblies in the other segments. Note that it refers to

PASS1, not PASS1F (and the same for 2 and 3). This is because PASS1F is the name of a ROM file, not an external DEF. Note also that "REF" has no DEFs, and therefore no official assembly name. It only has a file name.

9. One is going to have to put patches into the overlay areas, too (e.g., to branch out to the patch area). These patches have to be specified relative to an assembly name. But the patches can only refer to names that are present in the REF/DEF table for that segment. And any name that is REFed by the root will appear in the root's REF/DEF table and <u>not</u> in the overlay's table, which presents difficulties. We have just finished REFing all the assembly names (in paragraph 8), which means that none of them will be accessible to patches. To counteract this, all of the assemblies outside the root have had an extra DEF added at the front. This is the name of the assembly preceded by # (e.g.,, #STASCAN). Note that PASS1 is DEFed as #PASS1F, which, for consistency with the REFs in the root, should probably be #PASS1, since PASS1F is primarily with name of a file.

10. Finally, patches in the overlays have to be able to refer to the patch areas, which are in the root, and whose names are only in the root's REF/DEF table. To counteract this, an assembly is added to each of the overlays. In Pass 1, for example, the assembly is #PATCH1. It DEFs #PATCH1, which it EQUs to #PATCH, which it REFs. It also DEFs #PATCH1T, which it EQUs to #PATCHT, which it REFs. Thus there are now symbols available in the overlays that are equivalent to the symbols in the root that identify the patch areas.

    Like REF, the assemblies PATCH1, PATCH2, and PATCH3 are essentially no-size assemblies. They contain no useful data. They have all had one superfluous word of data added, because truly no-size object modules are not handled properly.

    We have seen that restrictions in the loading and/or patching processes cause considerable hassle in the processors. Even so, it might be possible to eliminate the six assemblies #PATCH, #PATCHT, #PATCH1, #PATCH2, #PATCH3, and REF by the following techniques: include #PATCH in PASS0 or MERRIMAC; include #PATCHT in FORTTEMP, include REF in PASS0 or MERRIMAC; and include the other three #PATCHn assemblies in PASS1, PASS2, and PASS3.

One of the effects of the above changes has been to increase the effective size of the compiler (i.e., the longest path) from 10,600 words to 13,000 words, an increase of more than 20%. Therefore, if later systems were to allow some of these changes to be undone (notably numbers 2, 4, 5, and 6), noticeable reductions in size might be obtained.

Xerox Corporation
701 South Aviation Boulevard
El Segundo, California 90245

XEROX

Reader Comment Form

We would appreciate your comments and suggestions for improving this publication.

Publication No.	Rev. Letter	Title	Current Date

How did you use this publication?

☐ Learning ☐ Installing ☐ Sales
☐ Reference ☐ Maintaining ☐ Operating

Is the material presented effectively?

☐ Fully Covered ☐ Well Illustrated ☐ Well Organized ☐ Clear

What is your overall rating of this publication?

☐ Very Good ☐ Fair ☐ Very Poor
☐ Good ☐ Poor

What is your occupation?

Your other comments may be entered here. Please be specific and give page, column, and line number references where applicable. To report errors, Please use the Xerox Software Improvement or Difficulty Report (1188) instead of this form.

Your Name & Return Address

2190(12/72)

Thank You For Your Interest. (fold & fasten as shown on back, no postage needed if mailed in U.S.A.)

Staple

Staple

Fold

First Class
Permit No. 229
El Segundo,
California

BUSINESS REPLY MAIL

No postage stamp necessary if mailed in the United States

Postage will be paid by

Xerox Corporation
701 South Aviation Boulevard
El Segundo, California 90245

Attn: Programming Publications

Fold

701 South Aviation Boulevard
El Segundo, California 90245
213 679-4511

XEROX

XEROX® is a trademark of XEROX CORPORATION.