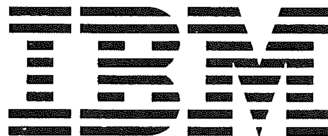
SC09-1130-01

C Compiler User's Guide For VM/CMS

IBM

SC09-1130-01

C Compiler User's Guide For VM/CMS



Second Edition (October 1987)

This edition applies to the IBM C for System/370 Program Offering (Products 5713-AAG and 5713-AAH).

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent program may be used instead.

This publication contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental

Publications are not stocked at the address given below. Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to

IBM Canada Ltd.
Information Development,
Department 849,
1150 Eglinton Ave East
North York, Ontario, Canada. M3C 1H7

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright Whitesmiths, Ltd. 1978, 1987
IBM C Compiler by Whitesmiths, Ltd.

IBM is a registered trademark of International Business Machines Corporation, Armonk, NY.

Contents

Chapter 1: C Compiler Overview

Introduction ....1 - 1
Compiler Architecture ....1 - 1
Linking ....1 - 1
Listings ....1 - 2

Chapter 2: Entering C Programs

Rules for C Program Source Files ....2 - 1
Entering Programs under CMS with XEDIT ....2 - 1

Chapter 3: Compiling Programs

Invoking the Compiler ....3 - 1
Compiler Options ....3 - 2
Using EXECs ....3 - 5

Chapter 4: Loading and Generating Programs

Using the Loader ....4 - 1
Using the GENMOD Command ....4 - 2
Linking Modules Not Produced By C ....4 - 3
Controlling Stack Size ....4 - 3
Reentrant Programs ....4 - 4
Creating Reentrant Programs ....4 - 4
How the Compiler Processes a Reentrant Program ....4 - 5
How RLINK Processes a Reentrant Program ....4 - 5
Using Discontiguous Saved Segments Under VM/CMS ....4 - 6
Defining a Named System for the DSS ....4 - 6
Compiling, Loading and Testing a Shared Segment ....4 - 6
Loading and Saving a Program into a Named Segment ....4 - 7
Attaching to the Named Segment ....4 - 7
Restrictions on Making Programs Reentrant ....4 - 10

Chapter 5: Executing Programs Under VM/CMS

Chapter 6: Debugging Programs

Run the Program to Termination ....6 - 2
Stepping One Source Line at a Time ....6 - 3
Stepping Through Multiple Source Lines ....6 - 3
Exit Debugging ....6 - 4
Examining Data Objects ....6 - 4
Displaying Current Source Line and File Name ....6 - 5
Executing Multiple Debugger Commands ....6 - 5
Logging Debugger Output ....6 - 6
Locating Source Code on Other Minidisks ....6 - 6
Setting Breakpoints ....6 - 6
Deleting Breakpoints ....6 - 7
More Debugger Commands ....6 - 7
Moving in Stack Frames ....6 - 7

Other Examine/Modify Commands ....6 – 8
Formatting Output from the Debugger ....6 – 8
Printing the Contents of Data Objects ....6 – 9
Type Casts ....6 – 9
Summary of Location Specifiers ....6 – 10

Chapter 7: Maintaining Libraries
Specifying Symbol Names ....7 – 1
Creating TXTLIB Files ....7 – 2
Adding a Member to a TXTLIB File ....7 – 2
Deleting a Member of a TXTLIB File ....7 – 2
Replacing a TXTLIB File Member ....7 – 2
Listing TXTLIB File Members ....7 – 2
Using Libraries ....7 – 3

Chapter 8: Interfaces with Other Languages
Calling Assembly Language Functions From C ....8 – 1
Mapping C Identifiers to External Names ....8 – 1
Transferring Control to Another Function ....8 – 2
Preserving Registers ....8 – 2
Passing Argument Data Objects ....8 – 3
Using the Stack ....8 – 4
Returning a Value ....8 – 5
Returning Control to the Calling Function ....8 – 5
Calling Functions Written in Other Languages from C ....8 – 5

Appendix A: Compiler Command Summary ....A – 1

Appendix B: Debugger Command Summary ....B – 1
Format Specifiers ....B – 2
Type Casts ....B – 2

Appendix C: EBCDIC Character Table ....C – 1

Index ....X – 1

Preface

The *C Compiler User's Guide for VM/CMS* (SC09-1130) is a reference guide for programmers writing C programs under VM/CMS on the System/370 architecture. It provides an overview of how the compiler works, and explains how to compile, link, execute, and debug programs using the System/370 C Compiler on VM/CMS. It assumes that you are familiar with VM/CMS.

You can find information about the C language and the C runtime library in your *C Language Manual* (SC09-1128).

Organization of this Manual

This manual is divided into eight chapters:

Chapter 1, "C Compiler Overview," describes the individual programs that make up the compiler and their interrelationship.

Chapter 2, "Entering C Programs," explains the process of creating a C source file under CMS and covers important compiler restrictions about line lengths and file characteristics.

Chapter 3, "Compiling Programs," describes the compilation process under CMS and on a VM batch machine, and how to take advantage of the many features and options available.

Chapter 4, "Loading and Generating Programs," explains how to use your linkage editor and loader to run programs under CMS and on a VM batch machine.

Chapter 5, "Executing Programs under VM/CMS," discusses how to run your program on CMS or on a VM batch machine. Procedures for passing data to a program are also described.

Chapter 6, "Debugging Programs," describes debugging techniques and the use of the C source level debugger included with the System/370 C compiler.

Chapter 7, "Maintaining Libraries," explains how to create TXTLIB files, and add or delete members of a TXTLIB file.

Chapter 8, "Interfaces with Other Languages," documents how to call a C function from within an assembly language routine, and how to write an assembly language routine that can be called as a C function.

Appendix A, "Compiler Command Summary," lists all the compiler options and what they do.

Appendix **B**, "Debugger Command Summary," lists all the debugger commands and command formats and what they do.

Appendix **C**, "EBCDIC Character Table," is a table of all EBCDIC characters and their hexadecimal values.

This manual also provides an **Index**.

Related Publications

The following publications are referred to in the text of this manual. You may want to use them for more information on certain topics.

* *C Language Manual*, SC09-1128.
* *VM/System Product CMS Primer*, SC24-5236.
* *VM/System Product Editor User's Guide*, SC24-5220.
* *VM/System Product CMS User's Guide*. SC19-6210.
* *VM/System Product Interpreter User's Guide*, SC24-5238.
* *VM/System Product EXEC2 Reference*, SC24-5219.
* *OS/VS-VM/370 Assembler Programmer's Guide*, GC33-4021.
* *VM/System Product CMS Command and Macro Reference*, SC19-6209.

Chapter 1: C Compiler Overview

This chapter explains how the compiler operates. The overview provides a basic understanding of the compiler architecture.

Introduction

The IBM C Compiler for System/370 is an integrated series of software development tools. It reads C source files and generates assembly language, which the assembler translates to object modules. The IBM linkage editor or loader combines object modules to produce executable files. You can request listings that show your C source interspersed with the assembly language that the compiler generates. You can also request that the compiler include a source level interactive debugger with your program.

You begin compilation by invoking the **CC** program. Under CMS, you provide **CC** with the name of your C source file. You can also specify various compiler options. If you plan to use the source level debugger with the program you are compiling, for example, you must specify the **DEBUG** option to **CC**.

When you invoke **CC** and provide it with the appropriate file name and options, the compiler generates an assembly language file, as well as any listings you request. A standard IBM assembler translates the assembly language file to an object module. You then invoke the IBM linkage editor or loader to combine the object modules to make an executable file.

Compiler Architecture

The compiler consists of several programs that work together to translate your C source files to assembly language files and listings. **CC** controls the operation of these programs automatically, using the information you provide in the options you specify to it.

The first step **CC** performs is to run the preprocessor. The preprocessor carries out preprocessor directives and expands macros. **CC** then runs two additional compiler passes to translate your preprocessed source code to an assembly language file. Finally, **CC** runs the IBM assembler you request to convert the assembly language file to an object module.

Linking

Once you have translated all of the separate compilations to object modules, you then run the IBM linkage editor or loader to combine all the object modules that make up your program with the

appropriate TXTLIB files from the C library. You can also build your own libraries and have the linkage editor or loader select files from them as well. The linkage editor or loader generates an executable file that you can invoke. If you specify the debugging option when you invoke **CC**, the linkage editor or loader will select the C library object modules that make up the source level interactive debugger and combine them with your program. When you run your program, any C source compiled with the debugging option will transfer control to the debugger before each executable statement.

Listings

You have several options for obtaining listings. If you request no listings, then **CC** error messages from each of the compiler passes are directed to your terminal, but no additional information is provided. Each error is labelled with the C source file and line number where the compiler detected the error.

If you request an assembler listing with interspersed C source, the compiler merges the C source as comments among the assembly language statements that it generates. Unless you specify otherwise, the error messages are still written to your terminal as well. **CC** runs a separate utility to produce the combined file which is then used as input to the assembler. Your listing is the listing output from the assembler.

If you request a listing of your C source with no assembly language listing output, **CC** gathers error messages from the three compiler passes and merges them with your C source to produce the listing. Unless you specify otherwise, the error messages are still written to your terminal as well. The same utility for merging the listing is used, but the combined file serves as input to a separate page formatting utility instead of the assembler. Your listing is the output from the page formatting utility.

Chapter 2: Entering C Programs

This chapter tells you how to enter C programs under VM/CMS, and describes the restrictions you need to consider in creating your C source file.

Rules for C Program Source Files

The C language imposes few restrictions on source files. You can place text in the first column of your file, unlike with some other programming languages. You can write C source files that use all of the columns in a text line. You can direct the compiler to concatenate a physical text line with the text line that follows by writing a backslash character at the end of the first physical text line. You must not write a C source file with a text line longer than 511 characters, however, either as a single physical text line or as two or more concatenated lines.

You create C source files using a subset of the EBCDIC character set. The C character set is extensive, so your input device may not be able to produce some of the characters you need to write your C program. The compiler lets you write special "trigraph" sequences to represent some characters. A trigraph is a series of three input characters which the compiler converts to the appropriate member of the C character set. You can replace the left brace, for example, with the trigraph `??<`. The replacement is permissible *anywhere* in your source code. You can also use the CMS **SET INPUT** and **SET OUTPUT** commands to map characters on your keyboard to the appropriate character you need for your C program.

For more information on the C character set and on trigraphs, see Chapter 2 "Elements of the C Language," in your *C Language Manual* (SC09-1128). More information on **SET INPUT** and **SET OUTPUT** is available in the *VM/System Product CMS User's Guide* (SC19-6210).

Entering Programs under CMS with XEDIT

Under CMS, you use the **XEDIT** editor to enter C programs. Use the **set case mixed** option to enter text in both lowercase and uppercase. You must write keywords (such as **while**) in lowercase, for example, but the type definition **FILE** must be in uppercase.

C is the file type for C source files. The default record format is **F** (fixed). The default logical record length is 80. The C

compiler accepts input in any record format and with any record length up to 511 characters. However, it will ignore columns 73 to 80 of a fixed record format file unless you specify the **NOSEQ** option to **CC**. Choose the logical record length that best suits your needs. A value of 80 for **lrecl** is probably a good choice most of the time. You may also wish to change the record format to **V** (variable) in many cases.

#include files may have any file type. **H** is the preferred type by convention.

Unless you specify otherwise, **XEDIT** sets the file mode to **A**.

For a complete introduction to **XEDIT**, read *VM/System Product CMS Primer* (SC24-5236), or *VM/System Product Editor User's Guide* (SC24-5220).

Chapter 3: Compiling Programs

This chapter tells you how to compile programs in a VM/CMS environment.

Invoking the Compiler

To invoke the compiler, type the command **CC**, followed by the name of the file you want to compile and any compiler options you choose. All the valid compiler options are described in this chapter. Commands to compile a program must have the form

```
CC <file name> [<file type> [<file mode>]] [(options]
```

Both the file type and file mode designations are optional. You can specify the file type and omit the file mode. However, if you specify the file mode you *must* specify the file type. The default file type is **C**. The default file mode is **A**. The option list is also optional.

The following example compiles the C source file **ECHO C A**. For a source listing of **ECHO**, see Chapter 6, "Debugging Programs."

```
CC ECHO
```

This command compiles and assembles your C program, creating the object module **ECHO TEXT A**.

Note: any file on the **A** minidisk that has the same file name as your program and a file type of **ASSEMBLE** will be lost during the compilation process.

If the compiler finds an error in your program, it will halt compilation before generating an object module. When an error occurs, the compiler sends a error message to your terminal screen, or to your listing if you request one. Appendix A of the *C Language Manual* (SC09-1128), "Compile Time Error Messages," lists the error messages the compiler generates and discusses possible causes of the error messages.

The example command above does not request any compiler options. In this case, the compiler will use only default information to compile and assemble your program. You can change the operation of the compiler by specifying the options you want when you run the compiler. To specify options to the compiler, type the appropriate option or options at the end of the command as shown in the first example above. Precede the option list with a left parenthesis. Separate the options you specify with spaces.

The part of the option name that appears in capital letters is the part of the name you *must* type for the compiler to interpret the option correctly. Specify at least that part of the option name. For example, **RE**, **REN**, and **RENT** all invoke the **REnt** option.

Compiler Options

(NO)ASM – When you specify **ASM**, the compiler produces an object module in the file *<filename>* **TEXT A**, where *<filename>* is the file name. When you specify **NOASM**, the compiler writes the assembler text to the file *<filename>* **ASSEMBLE A**, and does not generate an object module. If you specify **ASM** and the compiler detects an error, it will not invoke the assembler. The default is **ASM**.

CSect(name) – When you specify **CSECT**, the compiler produces a control section (CSECT) with the name *name*. When you specify the **RENT** option in combination with **CSECT**, the compiler also produces a second CSECT with the name $*name*. Therefore, *name* can be eight characters long when you specify **CSECT** alone, or seven characters when you specify **CSECT** in combination with **RENT**. The CSECT name you specify must not conflict with external function names or external data object names. See Chapter 8, "Interfaces with Other Languages," for information on the conversion of C names to object module symbol names.

(NO)DEBug – When you specify **DEBUG**, the compiler produces function calls that enter the debugger before each executable statement. Do not specify the **RENT** option in combination with **DEBUG**. If you specify **DEBUG** and **RENT** together the compiler generates an error message and halts. Automatic register allocation is disabled when you specify **DEBUG**. The default is **NODEBUG**, which produces no debugger calls.

Define(name=def,name=def, ...) – When you specify this option, the compiler defines one or more macros before it reads any of the source file you specify. Use this option to define symbols when you invoke the compiler instead of within your C source. Each definition has the form *name=def*. If you omit the *=def* part of the directive, the compiler assumes the definition to be "1". You may enter up to ten definitions, separated by commas. Defining a symbol in this way is equivalent to writing a preprocessor directive of the form

#define *name def*

in your C source. You may not define parameterized macros.

(NO)HASM – When you specify **HASM**, the compiler invokes the Assembler-H. When you specify **NOHASM**, it invokes the Assembler-XF. By default, the compiler uses the IBM assembler specified as the default assembler when the compiler was installed.

(NO)IDENT8 – When you specify **IDENT8**, the compiler restricts the number of significant characters in internal identifiers to 8 characters. This option provides support for C programs written for older compilers, where the programs depend on limited identifier significance. The default is **NOIDENT8** which restricts the number of significant characters in internal identifiers to 31 characters. Note that the compiler always restricts the length of external identifiers to eight characters for functions and seven characters for data objects.

LINecount(n) – This option specifies *n* as the number of lines you want the compiler to include in each page of a listing, including heading lines and blank lines. When you request a C source listing only, *n* can be any integer between 5 and 32767. When you request a listing that contains assembly language source, *n* can be any integer between 5 and 99 if you use Assembler–H or between 5 and 999 if you use Assembler–XF. See *OS/VS-VM/370 Assembler Programmer's Guide* (GC33-4021) for information on IBM assemblers. The default is **LINECOUNT(60)**.

(NO)LISt and (NO)SOurce – Listings help you review your code and error listings. You can use these two options separately or in combination to obtain the kind of listing you want. When you specify **LIST** alone, the compiler produces an assembly language listing, provided it detects no errors. If the compiler detects an error when **LIST** alone is specified, it produces a C source listing interspersed with error messages. When you specify **SOURCE** alone, the compiler produces a listing of C source code and error messages. When you specify both **SOURCE** and **LIST**, the compiler produces a listing of C source code interspersed with assembly language code, provided that it detects no errors. If the compiler detects an error when you specify both listing options, it produces a C source listing interspersed with error messages. If you specify two or more options which conflict, and are therefore invalid, the compiler generates an error message and halts. The following options are invalid in the combinations listed:

PPONLY and **LIST**
PPONLY and **SOURCE**
LIST and **NOASM**

The defaults are **NOLIST** and **NOSOURCE**, which produce no listings.

LSEarch(mod) – This option allows you to define a search modifier for **#include** files you enclose in double quotation marks. The search modifier *mod* specifies the minidisk you want the compiler to search first for your files. If the **#include** file is not on the minidisk you specify with *mod*, the compiler searches all subsequent minidisks in order. If the compiler does not find the **#include** file, it continues to search for it in the order you specify for **#include** files you enclose in angle brackets, as described under **SEARCH**

below. The default is to search for **#include** files in the order you specify for included files you enclose in angle brackets. For information on how the compiler searches for included files, see Chapter 7, "The Preprocessor," in your *C Language Manual* (SC09-1128).

(NO)MACarg – When you specify **MACARG**, the compiler expands macro argument identifiers within string constants, as the UNIX™ System V C compiler does. The default is **NOMACARG**, which does not expand string constants.

(NO)PPonly – When you specify **PPONLY**, the compiler runs only the preprocessor. This expands all macros and shows all include files in the C source, without compiling the source. The compiler writes the expanded source code to <*filename*> **EXPANDED A** under CMS, where <*filename*> is the file name. The default is **NOPPONLY**, which compiles your source to an object module. Note that the preprocessor does not perform the same error checking when you specify **PPONLY** as it does when you compile your source to an object module.

(NO)REnt – When you specify **RENT**, the compiler produces reentrant code by making every access to a data object through a base register address. Data objects can therefore be placed at arbitrary locations in memory. You must specify the **CSECT** option in combination with **RENT**. Do not specify the **DEBUG** option in combination with **RENT**. If you specify **RENT** and **DEBUG** together the compiler generates an error message and halts. The default is **NORENT**, which does not produce reentrant code unless your program contains no global or static data objects. Chapter 4, "Loading and Generating Programs," tells you how to make your program reentrant.

SEarch(mod) – This option allows you to define a search modifier for preprocessor **#include** files that you enclose in angle brackets **< >** within your program. *mod* represents the minidisk the compiler will search first for the **#include** files. If the **#include** file is not on that minidisk, it will search all subsequent minidisks in order. The compiler will search all accessed minidisks in normal search order if you omit the search modifier. For more information on how the compiler searches for **#include** files, see Chapter 7, "The Preprocessor," in your *C Language Manual* (SC09-1128).

SEQ(m,n)|NOSEQ – When you specify **SEQ(*m,n*)**, the compiler ignores columns *m* through *n*, inclusive, in each line of your C source. The default for variable length record format files is **NOSEQ**, which does not ignore any columns. The default for fixed record format files is **SEQ(73,80)**. If you specify **NOSEQ** in combination with **SEQ(*m,n*)**, the compiler generates an error message and halts. If you specify an invalid range of columns to ignore, the compiler generates an error message and halts.

UNIX is a registered trademark of AT&T Bell Laboratories

(NO)SHowinc – When you specify **SHOWINC**, the compiler shows the contents of **#include** files in listings. When you specify **NOSHOWINC**, the compiler does not show the contents of included files. The default is **NOSHOWINC**.

(NO)TErminal – When you specify **NOTERMINAL**, the compiler writes no output to your terminal screen. When **NOTERMINAL** is specified, the compiler sends error messages to the file *<filename>* **LISTING A**, where *<filename>* is the file name. If you request a listing, error messages are sent only to the listing file. The default is **TERMINAL**, which writes error messages to your terminal screen.

(NO)UPCONV – When you specify **UPCONV**, the compiler uses "unsignedness preserving" rules for type conversions. You use this option to compile older C programs that may depend on the older conversion rules. The default is **NOUPCONV**, which means that the compiler uses the "value preserving" rules adopted in the proposed ANSI standard for C.

The following example compiles the C source file **MYPROG C A**, invokes Assembler-H, and searches minidisk H first when including **#include** files:

```
CC MYPROG (HASM SEARCH(H)
```

Using EXECs

Use EXECs when you want to perform a compilation that requires a number of commands that are time consuming to retype. You can also use EXECs to compile programs on a VM batch machine.

It is recommended that you use **EXEC2** or **REXX** rather than the older **EXEC** interpreter if you are compiling a C program using EXECs. For more information on **EXEC2**, refer to the *VM/System Product CMS User's Guide* (SC19-6210), or the *VM/System Product EXEC2 Reference* (SC24-5219). For more information on REXX, refer to the *VM/System Product Interpreter User's Guide* (SC24-5238).

begins with the first instruction of the first object module loaded. All object modules that the C compiler produces begin with a code segment that initializes the C environment properly. You can therefore specify the files in any order. If you need to specify more files than you can fit on the **LOAD** command line, use the **INCLUDE** command in conjunction with **LOAD**. Always specify the * option to **START**.

If you reference functions or external data in your C source files without defining them, the loader will search for them in the library or libraries you designate using the **GLOBAL TXTLIB** command. If any external references in the object modules are still undefined, the loader issues the error message "THE FOLLOWING NAMES ARE UNDEFINED" followed by a list of unsatisfied references. However, it loads your object module anyway. Your program may not run as you expect if this happens. Note also that the names that the loader issues do not exactly match the names in your C source. See Chapter 8, "Interfaces With Other Languages," for an explanation of the conversion of names in a C program to names within object modules.

If your accessed minidisks contain a file with the name "*<NAME>* **TEXT**," where *<NAME>* is the name of an external function or data object you reference in your C source program, the loader will include this file when it loads your program. This is because it searches the accessed minidisks before searching the library you specify with the **GLOBAL TXTLIB** command.

In addition to loading the program, the **LOAD** command creates the file **LOAD MAP** on disk **A**. This file contains the load addresses of all symbols defined in your program. It is quite useful for program debugging.

Note: The entry point for a C program is **C$START**. You do not need to specify an entry point for most programs. The object modules that the C compiler produces will use **C$START** as the entry point automatically.

For information on the **GLOBAL**, **INCLUDE**, **START** and **LOAD** commands, refer to *VM/System Product CMS Command and Macro Reference* (SC19-6209).

Using the GENMOD Command

The **LOAD** command creates an executable program in memory. You destroy this executable program as soon as you issue another **LOAD** command. You can save the program as a nonrelocatable module using the **GENMOD** command. You can run this nonrelocatable module directly by typing its name. You must run **GENMOD** immediately after you load your object module using the **LOAD** command.

The example below shows a typical series of commands to create and run an executable module:

```
LOAD ECHO
GENMOD ECHO
```

```
ECHO HELLO, WORLD!
```

The **LOAD** command loads **ECHO TEXT A.** **GENMOD** creates the nonrelocatable executable module **ECHO MODULE A**, which you can run by name. The strings "HELLO," and "WORLD!" are arguments to **ECHO**.

Read *VM/System Product CMS Command and Macro Reference* (SC19-6209) for more information on the GENMOD command.

Linking Modules Not Produced By C

The **LOAD** and **GENMOD** command sequences, as described above, rely on default values for the entry point address and starting storage location of your program. If you link object modules that the C compiler produces with object modules that are *not* produced by the C compiler, these default values will be incorrect and the results will be unpredictable. To avoid this problem, you must specify **LOAD** and **GENMOD** commands that designate an explicit entry point address and starting storage location. You meet these requirements by loading a special object module, **C$TEXT TEXT**, which comes with the C compiler. The following example loads **C$TEXT TEXT** with the program **MYPROG**:

```
LOAD C$TEXT MYPROG (RESET C$START
GENMOD MYPROG (FROM C$TEXT
```

You must specify **C$TEXT** as the first module on the **LOAD** command line. It contains the correct initialization code for C programs. The **RESET C$START** option to **LOAD** specifies that **C$START** is the program entry point. By specifying **C$TEXT** first in the list of object modules and by specifying **FROM C$TEXT**, you also guarantee that **GENMOD** will use **C$TEXT** as the starting storage location. You must *always* use this command sequence when linking modules not produced by C. You may also use it when linking only C modules.

Controlling Stack Size

Your program allocates calling environments and data objects with dynamic lifetime within a storage area called a "stack." By default, the stack occupies 50,000 bytes. If your program terminates abnormally because the signal **SIGSTACK** was reported, it requires a larger stack.

A "stack overflow" is often the result of a program error. If you call a function with uncontrolled recursion, your program will eventually run out of stack. If you declare very large arrays with storage class **auto** or **register**, your program may report a stack overflow when the block that declares these large data objects is entered. Do not increase stack size to overcome a program error.

If your program requires a larger stack, you can change the stack size. To increase stack size (to 80,000 bytes in this example), add a file level definition of the form

```
const int _stksize = 80000;        /*  for 80,000 byte stack  */
```

to any one of the compilations making up your program. Recompile and relink the program, and it will run with the stack size you specify as the value of **_stksize**.

Reentrant Programs

Making a program reentrant can significantly decrease system memory usage if the program is frequently run by more than one user simultaneously. When multiple invocations of a reentrant program run, each user has an individual copy of the data objects, but all share a single copy of the program instructions. Do not make a program reentrant that is seldom run by more than one user at a time. A single invocation of a reentrant program requires more memory than an identical program that is not reentrant.

Creating Reentrant Programs

A C program that does not modify static data objects is reentrant without compiling or linking for reentrancy. Other C programs can be made reentrant. Restrictions concerning reentrant programs are discussed below. You compile a program for reentrancy by specifying the **RENT** and **CSECT(*name*)** options to **CC**. When you specify the **RENT** option, the compiler compiles your program so that data objects can by accessed relative to a base address register. Data objects can therefore be placed at arbitrary memory locations. When you specify the **CSECT(*name*)** option, the compiler creates a control section (CSECT) and external dummy section (DXD) used to hold static data objects. These processes are described below.

After you compile a program for reentrancy you must process it using the **RLINK** program before creating an executable module. **RLINK** processes the external dummy sections the compiler creates and provides initialization data as described below. The output of **RLINK** is a single reentrant object module which you use as input to the linkage editor or loader. Appendix A, "Compile Time Error Messages," in your *C Language Manual* (SC09-1128) discusses the error messages that **RLINK** generates.

To invoke **RLINK**, type the command **RLINK** followed by the names of one or more object modules. Commands to process reentrant modules prior to loading must have the form

```
RLINK <object module>
```

After you run **RLINK** you run the linkage editor or loader to create an executable module. On VM/CMS, you must define a reentrant program as a named discontiguous saved segment in the CP nucleus, so that all virtual machines can use it. This process is described below.

How the Compiler Processes a Reentrant Program

In reentrant programs, the compiler associates a pair of external identifiers with all non "const" data objects. When compiling the program, the compiler truncates external identifiers to seven characters. To create the first member of the pair, the compiler prepends a '$' character to the truncated identifer. This identifier names an external dummy section. To create the second member, the compiler prepends a '#' character to the truncated identifier. This identifier names a constant that holds the initial value of the data object. For example, a data object you declare as

```
int maxval = {1000}
```

has associated with it the assembly language statements **$MAXVAL DXD 4C** and **#MAXVAL DC F'1000'**.

The compiler writes static data objects into the external dummy section $<*csect*> and into the CSECT #<*csect*>, where <*csect*> is the name you provide when you specify the **CSECT(*name*)** option to **CC**. No global function or data object may have the same name as a CSECT name you specify.

How RLINK Processes a Reentrant Program

RLINK takes as input one or more object modules and an optional TXTLIB and produces as output a single object module.

RLINK first searches the TXTLIB you specify to satisfy external references. Note that **RLINK** will not generate an error message if it encounters references it cannot resolve. These references will remain unresolved in the output object module. The FILEDEF **INCLUDE** can specify a TXTLIB whose members have been compiled for reentrancy. See Chapter 7, "Maintaining Libraries," for information on including object module libraries with your C program.

RLINK then collects external dummy sections. It calculates the size and offset of each dummy section, resolves all references to it, and calculates its total size. **RLINK** does not leave any information about external dummy references in the object modules it processes.

Next, **RLINK** creates the control section **C#INIT1**, which contains initialization information that the C runtime uses to initialize the allocated data section. If you do not run **RLINK** to process your program, the linkage editor or loader will use the dummy table **C#INIT1** that the C library defines. Because the C library defines a dummy version of this table when you do not specify the **RENT** option, do not specify the C library as input to **RLINK**. For information on the C runtime, see Chapter 8, "C Runtime Environment," in your *C Language Manual* (SC09-1128).

You pass input files to **RLINK** as arguments. On VM/CMS, you *must* specify one or more input files. **RLINK** sends its output to **RLOBJ TEXT *** under VM/CMS.

For example, to compile, preload, load and run the C program in files **PROG1 C A** and **PROG2 C A** and the TXTLIB **'PROGLIB TXTLIB**

A' as a reentrant program under VM/CMS, write:

```
CC PROG1 ( RENT CSECT(PRO1)
CC PROG2 ( RENT CSECT(PRO2)
FILEDEF INCLUDE DISK PROGLIB TXTLIB A
RLINK PROG1 PROG2
GLOBAL TXTLIB CLIB
LOAD C$TEXT RLOBJ ( RESET C$START
GENMOD PROG (FROM C$TEXT
```

Using Discontiguous Saved Segments under VM/CMS

To run a reentrant program under VM/CMS, you must define it as a named discontiguous saved segment in the CP nucleus.

Defining a Named System for the Discontiguous Saved Segment

You must define the name, location on disk, load address and size of a discontiguous saved segment. You do this by defining the **NAMESYS** macro in the CP system name table file **DMKSNT ASSEMBLE**. Before a discontiguous saved segment can be attached by name, it must be loaded and saved. It must be loaded at an address that is beyond the highest address of any virtual machine that will attach to it. Your system administrator must add the named system to the CP nucleus for the discontiguous shared segment.

Below is a sample entry for a system named **TSTSYS**. **TSTSYS** starts at address 600000 hex and is 256 kbytes in size.

```
***********************************************************************
*  TSTSYS                                                             *
*  HEX LOAD ADDRESS FOR SEGMENT  96 = 600000                          *
*  THE SPACE FOR TSTSYS IS ALLOCATED ON VMSEXT, AS FOLLOWS:           *
*  (THE ALLOCATION IS BASED ON 24 PAGES/3340 CYLINDER;                *
*        1 SEGMENT = 16 PAGES, 1 PAGE = 4 KBYTES)                      *
*  CYL 312, PAGE 01 TO CYL 314 PAGE 17 (65 PAGES)                     *
***********************************************************************
         SPACE
TSTSYS   NAMESYS SYSNAME=TSTSYS,                                      X
               SYSSIZE=256K,                                          X
               SYSVOL=VMSEXT,                                         X
               SYSCYL=,                                               X
               SYSSTRT=(312,01),                                      X
               SYSPGCT=64,                                            X
               SYSPGNM=(1536-1599),                                   X
               SYSHRSG=(96,97,98,99)                                  X
```

Compiling, Loading and Testing a Shared Segment

Perform the following steps to create and test the reentrant program **ECHO C A**:

```
CC ECHO ( RENT CSECT(ECHO)
RLINK ECHO
```

Loading and Saving a Program into a Named Segment

You must save a discontiguous saved segment so that all virtual machines can use it. To load and save a reentrant program into a named segment, perform the following steps:

1. Expand the virtual memory of your machine by entering the command

   ```
   DEFINE STORAGE 8M
   ```

   Your system administrator must give you permission to expand the size of your virtual machine to eight megabytes.

2. Run the Initial Program Loader to start VM/CMS on the larger virtual machine by entering the command

   ```
   IPL CMS PARM AUTOCR
   ```

3. Load the **ECHO** program at address 600000 hex by entering the commands

   ```
   GLOBAL TXTLIB CLIB
   LOAD RLOBJ (ORIGIN 600000
   ```

4. Set the storage keys by entering the command

   ```
   SETKEY 13 TSTSYS
   ```

5. Save the system by entering the command

   ```
   SAVESYS TSTSYS
   ```

   You must have privilege class **E** to perform a **SAVESYS** operation.

Attaching to the Named Segment

A source code listing of the assembly language program **RUNSYS** appears below. **RUNSYS** loads and executes a saved segment.

```
*         STARTUP OF DISCONTIGUOUS SAVED SEGMENT IN VM/SP
*
RUNSYS    CSECT
          STM      14,12,12(13)
          LR       2,0
          LR       3,1
          LR       12,15
          USING    RUNSYS,12
          ST       13,SAVE+4
          LR       1,13
          LA       13,SAVE
          ST       13,8(1)
*
*         FIND SAVED SEGMENT
*
          LR       4,3              SEGMENT NAME
          LA       5,X'0C'          FINDSYS
          DC       X'83'            DIAGNOSE
          DC       X'45'            RX = 4, RY = 5
```

```
         DC       X'0064'
         BC       2,NOSUC         CC = 2, ERROR
         BC       8,RUNSYS        CC = 0, ALREADY LOADED
*
*        GET VIRTUAL MACHINE SIZE
*
         DC       X'83'           DIAGNOSE
         DC       X'67'           RX = 6, RY = 7
         DC       X'0060'
         CLR      6,4             SEGMENT ABOVE VIRTUAL MACHINE ?
         BL       LOAD            YES, LOAD IT
         LA       1,ERRMSG1
         LA       5,178           RETURN CODE
         B        ERROR
*
*        LOAD SAVED SEGMENT
*
LOAD     LR       4,3             SEGMENT NAME
         LA       5,0             LOADSYS IN SHARED MODE
         DC       X'83'           DIAGNOSE
         DC       X'45'           RX = 4, RY = 5
         DC       X'0064'
         BC       2,NOSUC         CC = 2, ERROR
*
*        RUN SAVED SEGMENT
*
RUN      LR       0,2
         LR       1,3
         LR       15,4
         BALR     14,15
*
*        PURGE SAVED SEGMENT
*

         LR       4,3             SEGMENT NAME
         LA       5,8             PURGESYS
         DC       X'83'           DIAGNOSE
         DC       X'45'           RX = 4, RY = 5
         DC       X'0064'
         BC       2,NOSUC         CC = 2, ERROR
*
*        RESTORE SAVE AREA
*
RETURN   L        13,SAVE+4
         ST       15,16(13)
         LM       14,12,12(13)
         BR       14
*
*        DETERMINE RETURN CODE
*
NOSUC    LA       4,44
         CR       4,5
         BNE      NOSUC1
         LA       1,ERRMSG2
```

```
         B        ERROR
NOSUC1   LA       4,177
         CR       4,5
         BNE      NOSUC2
         LA       1,ERRMSG3
         B        ERROR
NOSUC2   LA       1,ERRMSG4
*
*        WRITE ERROR MESSAGE
*
ERROR    LH       0,0(1)         MSG LENGTH
         LA       1,2(1)         MSG
         SVC      93             TPUT
         LR       15,5           RETURN CODE
         B        RETURN
*
*        ERROR MESSAGES
*
ERRMSG1  DC       H'21'
         DC       C'SEGMENT ADDRESS ERROR'
*
ERRMSG2  DC       H'28'
         DC       C'NAMED SEGMENT DOES NOT EXIST'
*
ERRMSG3  DC       H'16'
         DC       C'PAGING I/O ERROR'
*
ERRMSG4  DC       H'5'
         DC       C'ERROR'
*
*        SAVE AREA
*
SAVE     DS       18F
*
         END
```

Examples

The example below loads and executes the **RUNSYS** program. **RUNSYS** uses the name by which you invoke it as the name of the saved segment to attach. In the example, **TSTSYS** is the name by which **RUNSYS** is invoked. The load module it generates therefore has the name **TSTSYS** also.

1. To reset the size of the virtual machine, enter the commands

```
DEF STOR 1M
IPL CMS PARM AUTOCR
```

2. To assemble and rename **RUNSYS**, enter the commands

```
ASSEMBLE RUNSYS
RENAME RUNSYS TEXT A TSTSYS = =
```

3. To load **TSTSYS** and generate an executable module, enter the commands

```
LOAD TSTSYS
GENMOD TSTSYS
```

To run the **ECHO** program as a discontiguous saved segment named **TSTSYS** and pass it the arguments "**HELLO,** " and "**WORLD!**", enter the command

```
TSTSYS HELLO, WORLD!
```

Restrictions on Making Programs Reentrant

The current version of IBM C for System/370 does not support the initialization of static data objects (data objects with storage class **extern** or **static**) with the address of a non "const" static data object within reentrant programs. In addition, you must supply all the information concerning a data object with storage class **static** the first time you declare the data object. The current version does not support tentative declarations in a reentrant program, even to the restricted extent supported by the ANSI Draft Proposed Standard for C of April, 1985.

Chapter 5: Executing Programs under VM/CMS

You execute an object module by first issuing the **LOAD** command, which loads the object module into storage. Then issue the **START** command. See Chapter 4, "Loading and Generating Programs," for information on the **LOAD** command.

The example below loads and executes the **ECHO TEXT** object module. For a source code listing of the C program **ECHO**, see Chapter 6, "Debugging Programs." In this example, **abc** and **def** are the arguments to the **ECHO** program.

```
LOAD ECHO
START * abc def
```

When you use the **START** command, control is passed to your program.

If you specify arguments to your program when you run the **START** command, you must specify the entry point as *. Otherwise, the **START** command will interpret the first of your program arguments as an entry point specification. If your program does not require any arguments, you may omit the * entry point specification.

To execute a load module, enter your module name, followed by the arguments to the program. The example below illustrates how to run the load module **ECHO MODULE.** For a source listing of the **ECHO** program, see Chapter 6, "Debugging Programs."

```
ECHO abc def
```

When executing your program under VM/CMS, the three standard text streams are connected to your terminal. **stdout** and **stderr** are open for writing to the terminal and **stdin** is open for reading from the terminal.

On VM/CMS, **argc** has a value equal to the number of command arguments you specify plus the program name, and **argv[0]** points at a string giving the program name. In the example above, **argc** has the value 3 and **argv[0]** points at the string "ECHO".

Read the *CMS User's Guide* (SC19-6210) for more information on entry points and on executing programs. For more information on redirecting input to and output from your program, specifying multiple command line arguments, and other aspects of command line processing, see your *C Language Manual* (SC09-1128).

Chapter 6: Debugging Programs

The C source level debugger provides a controlled environment for program execution and examination. You invoke the debugger by running an executable file that you compile with the **DEBUG** option. The linkage editor or loader links the debugger components with your program.

Once you enter the debugger (by executing a statement that was compiled with debugging enabled), you can request a variety of services. You can display a particular data object when its stored value changes. You can stop execution when the debugger reaches a particular C source line. You can display each source line of your program before it executes. You can answer questions about the values stored in data objects, or send a copy of your debugger session to a file.

The following examples demonstrate the debugging of the program **ECHO C A**. To follow along with this manual, enter the C source code for the **ECHO** program and compile **ECHO** with debugging enabled. The examples step you through the control commands you use most often in a debugging session.

The source code for the example program **ECHO** is:

```
1    /*  ECHO ARGUMENTS TO STDOUT
2     */
3    #include <stdio.h>
4
5    int main(ac, av)
6        int  ac;
7        char *av[];
8        {
9        ++av, --ac;
10       for (; 0 < ac; ++av, --ac)
11           {
12           printf("%s", *av);
13           if (1 < ac)
14               printf(" ");
15           else
16               printf("\n");
17           }
18       exit(0);
19       }
```

To compile **ECHO C A** enter:

```
CC ECHO (DEBUG
```

The compiled program is **ECHO TEXT A**. The sample program runs without any errors. Note that the executable and source files are located on the same minidisk and have the same file name. This simplifies debugging **ECHO**, because the executable file takes the source file and displays it on the screen during the debugging session. If these two files are not on the same minidisk, you must provide the search path the debugger needs to find the source file. The **f** command specifies the search path, as described below. To run **ECHO**, enter:

```
GLOBAL TXTLIB CLIB
LOAD ECHO
START * abc def
```

The debugger will display

```
| main():5    |int main(ac, av)
```

on the screen (assuming the source to **ECHO** is on the current minidisk). The first part of the output line, between the vertical bars, tells you the name and source line number of the function in which you are currently executing. The second part, after the second vertical bar, is the contents of the source line.

The symbol **:>** that appears on your screen is the debugger prompt. When you see the **:>** prompt, it means that the source level debugger is awaiting a command. Type debugger commands only in response to the **:>** prompt.

When you use the debugger, you enter one of two types of commands:

1. a command that controls the flow of execution
2. a command that examines or modifies a location.

All commands are in the following format:

<location> <cmd letter> <args>

<location> specifies an optional location within the program, a data object, or an unsigned decimal integer. *<cmd letter>* specifies a debugger command. *<args>* specifies an optional argument to the command. If you specify a location, you must separate it from the command itself by spaces or horizontal tabs. The debugger interprets *<location>* and *<args>* differently, depending on the specifics of a given command.

Run the Program to Termination

The simplest debugger action is to run the entire program to termination. Type g in response to the **:>** prompt. The **ECHO** program then generates the output:

```
abc def
```

The **g** command tells the debugger to continue to the next "breakpoint." A breakpoint is a stopping place that you specify

with a debugger command, as explained below. Since there are no breakpoints in this example, the program runs to termination.

Stepping One Source Line at a Time

A second option is to run the program by "single stepping" through the entire run one source line at a time. To rerun **ECHO TEXT**, enter:

```
LOAD ECHO
START * abc def
```

The debugger will display:

```
| main():5    |int main(ac, av)
```

This time, press the <ENTER> key once after each output line the debugger displays. The result is:

```
    | main():9     |        ++av, --ac;
    | main():10    |        for (; 0< ac; ++av, --ac)
    | main():12    |            printf("%s", *av);
    | main():13    |        if (1 < ac)
    | main():14    |            printf(" ");
    | main():10    |        for (; 0< ac; ++av, --ac)
    | main():12    |            printf("%s", *av);
    | main():13    |            if (1 < ac)
    | main():15    |            else
    | main():16    |              printf("\n");
abc def
    | main():10    |        for (; 0< ac; ++av, --ac)
    | main():18    |        exit(0);
```

Note that the debugger displays only those source lines that are executed. It does not display preprocessor directives, function definitions, or other declarations. Also note the output line `abc def` after the execution of line 16. This is the *actual* output of the **ECHO** program. The vertical bars help you distinguish regular program output from debugger messages.

Stepping Through Multiple Source Lines

Normally, pressing <ENTER> steps through the program just one line at a time. You can step more than one line by writing a repeat count in front of the **s** command. For example, the debugger command **10 s** steps through your program ten source lines at a time. To try this feature, run **ECHO** and type 500 s in response to the **:>** prompt. If you instruct the debugger to step through more source lines than there are lines to execute, the program runs to termination.

If you do not want to see the output of the debugger as it steps through the source lines, write a repeat count in front of the **g** command. Rerun **ECHO** and type 5 g in response to the **:>** prompt. The debugger will display:

```
| main():5   |int main(ac, av)
| main():14  |              printf(" ");
```

Exit Debugging

To end a debugging session without stepping through to program termination, simply enter q in response to the :> prompt.

Examining Data Objects

Another option available to you through the source level debugger is to observe what happens to the data objects in your program during program execution. To see how this works, rerun **ECHO** again by entering:

```
LOAD ECHO
START * abc def
```

The program will display:

```
| main():5   |int main(ac, av)
```

Type v in response to the :> prompt. The following list appears:

```
variables in main()
argument signed char **av
argument signed int   ac
```

If there are global data objects in your program, the debugger lists them, under the heading variables in echo. This list includes all the data objects visible to the current function, as defined by the scope and visibility rules of C. Note that the debugger then prompts you for input again. It stops executing source lines and waits for further instructions. Unless you type g or press only <ENTER>, it will continue to wait for additional instructions.

You can also ask the debugger to print the values stored in data objects. In this example, the simplest data object near the top of the list is **ac**. It is the count of the number of command options you supply to **ECHO**. Type :ac p in response to the :> prompt to access and print the stored value. The space after the data object name indicates the end of the data object name and the beginning of the "print" command.

```
argument signed int   ac      value: 3
```

There are three command options to **ECHO**: ECHO (the program name), abc, and def. The name of the program always counts as a command option. In this example, watch how **ac** changes. Press <ENTER> two times to execute the next two lines. The debugger will display:

```
|main():10     |    for (; 0 < ac; ++av, --ac)
| main():12  |              printf("%s", *av);
```

The program adjusts the value stored in the data object **ac** to track the number of command options that have yet to be processed. Type :ac p in response to the :> prompt to print out the value stored in the data object **ac**. The debugger will display:

```
argument signed int    ac       value: 2
```

To print **av**, of type *pointer to pointer to char*, type :av p in response to the **:>** prompt. The debugger will display:

```
argument signed char   **av      value: 31274
```

Note: The actual values for **av** *and* ***av** *depend on where the program is loaded into memory.*

You can specify two levels of indirection to print the character pointed to. Type :**av p%d in response to the **:>** prompt. The debugger will display:

```
argument signed char   **av      value: 31274->31328->129
```

You can display this character as an EBCDIC character. Type :**av p%c in response to the **:>** prompt to print out the character. The debugger will display:

```
argument signed char   **av      value: 31274->31328->a
```

You can also print out the *pointer to char*. Type :*av p in response to the **:>** prompt. The debugger will display:

```
argument signed char   **av      value: 31274->31328
```

You can even use a *pointer to char* to print out a string. Type :*av p%s in response to the **:>** prompt and the value of the second command line option is displayed.

```
argument signed char   **av      value: 31274->abc
```

Displaying Current Source Line and File Name

You can display the current source line and file name during the debugging session. Type a period . in response to the **:>** prompt to view the current source line. Press <ENTER> to go to the next source line and then type f in response to the **:>** prompt to display the current file name.

Assuming that you are at line "main( ):10" when you issue the command, the debugger will display:

```
 |main():10      |    for (; 0 < ac; ++av, --ac)
file: echo c a
```

The file name displayed is that of the original C source file, **ECHO C A**.

Executing Multiple Debugger Commands

The debugger allows you to string several commands together. Single step to source line 10 and type :ac p;:*av p%s in response to the **:>** prompt to display the number of options and the contents of the current option.

```
|main():10   |    for (; 0 < ac; ++av, --ac)
argument signed int   ac      value: 1
argument signed char   av      value: 31276->def
```

Note that the debugger breaks up multiple commands on an input line and repeats the ":*av p%s" part on a line by itself.

To specify multiple commands on a single input line (in response to a single **:>** prompt), separate the commands with semicolons. However, certain commands take the remainder of the input line following the one character command name as an argument, regardless of what is actually present on the command line. You cannot follow such a command with other commands. The breakpoint command **b**, for example, interprets the remainder of the input line as actions to be taken if control is transferred to the specified location. The **f** command also ignores the remainder of the input line. The **w**, **>**, and **<** commands interpret the remainder of the input line as a file name. Do not specify other debugger commands in conjunction with these commands.

Logging Debugger Output

You can write a file containing a log of all debugger output. For example, run **ECHO** and type w log 1 in response to the **:>** prompt. Here, **w** is the command to write debugger output to a file, and **log 1** is the file name and file mode of the file where the debugger stores the output. Specification of the file type is optional. Both parts of the command are necessary to create a log file. Single step down a few source lines, and then type w in response to the **:>** prompt. Move down a few more source lines and type q in response to the **:>** prompt. Display the contents of the file **log 1** on your terminal screen as you would the contents of any other text file on your system. Note that it shows only debugger input and output. The debugger does not copy ordinary output from **ECHO** to the log file.

Locating Source Code on Other Minidisks

The **f** command specifies the file name the debugger must use to look up source lines. Follow the **f** command with a space and an alternate minidisk name, so the debugger can locate the source file if it is not on the current disk.

Setting Breakpoints

The **b** command sets or displays "breakpoints." A breakpoint is a condition that you define. You can set a breakpoint for when the value stored in a data object changes, or for when control passes to a particular source line. To see how you use breakpoints, type :ac b in response to the **:>** prompt. This command sets a breakpoint for changes in the value stored in **ac**. The debugger will display:

```
set at argument signed  int  ac  []
```

The response [] shows that you did not request any specific action. The default action upon reaching a breakpoint is to wait for further debugger commands from the keyboard. Note that if you wish to step through source lines when the debugger reaches a breakpoint, you must reenter the **s** command. Type g in response

to the :> prompt. Execution resumes and continues until the program reaches the source line "main( ):10." The debugger will display:

```
  at 31268 argument signed  int   ac modified
     |main():10      |    for (; 0 < ac; ++av, --ac)
```

To set a breakpoint at source line 12 in the function, type main():12 b in response to the :> prompt. The debugger will display:

```
set at main():12        []
```

To list all the breakpoints currently set, type b in response to the :> prompt. The debugger will display:

```
 main():12        []
argument signed  int ac  []
```

One use for setting breakpoints is to help you track what is happening to a given data object. To see how to track a data object, type g in response to the :> prompt to execute the program until you encounter a step where the program modifies the value stored in the data object **ac**. The debugger will display:

```
| at 31268 argument signed  int  ac modified
     |main():10      |    for (; 0 < ac; ++av, --ac)
```

You can then continue execution until you encounter another source line that alters the value stored in the data object **ac**.

Deleting Breakpoints

The **d** command deletes specified breakpoints. If you associate several breakpoints with the same data object, the **d** command will delete the first one set. To delete other breakpoints associated with the same data object, repeat the same **d** command as you used to delete the first breakpoint set. This will delete one breakpoint at a time beginning with those you set first. The last breakpoint remaining will be the one set most recently. Type :ac d in response to the :> prompt to delete the breakpoint associated with **ac**.
The debugger will display:

```
reset argument  int  ac  []
```

More Debugger Commands

The following is a list of additional source level debugger commands that you can use. To try each one, perform the steps listed beneath its description.

Moving in Stack Frames

There is an additional command you can use to change the scope you are inspecting with the debugger. Write **m** followed by a direction option to move up or down "stack frames" in the direction you specify. Stack frames are the regions of storage that

the compiler allocates and deallocates from the region of storage known as the "stack." A stack frame holds the calling environment of the expression that called the function, the argument data objects passed on the function call, and all of the data objects declared within the function that have dynamic lifetime. You direct movement of the scope with a **u** to move up one stack frame, **d** to move down one stack frame, **t** to move to the top stack frame, and **b** to move to the bottom stack frame. The function **main** is at the top of the stack. You can precede your **m** command with a location specifier. If you specify a location, the debugger moves as many times as necessary to get to a stack frame that is in scope for that location.

Other Examine/Modify Commands

The **t** command displays a complete list of known stack frames from the current stack frame to the top stack frame (usually **main()**). You can specify the option **v** to find the data object names and associated type information for each frame. If you type tv&, the debugger also prints the address of each data object.

The **u** command updates a data object you specify by storing a new value in it. If you do not specify a value, the debugger prompts you for one. The debugger interprets this value according to the format specification you used most recently to print the value stored in the data object. You can use an = in place of a **u**. Both commands have the same effect. You can only apply these commands to location specifiers.

The **v** command lists the name and type information for all data objects the debugger knows of in the current stack frame. If you specify **&**, the debugger prints the address of each data object.

A **w** command writes a log of all debugger input and output to the file name you specify. Make sure a file by the same name as the one you specify does not exist. The log begins immediately. If you do not specify a file, the debugger stops writing to any previous log file.

You can redirect debugger input and output as well. Use a >, followed by a file name, to redirect debugger command output and prompts to the named file. Use a <, followed by a file name, to redirect debugger command input to come from the named file.

Formatting Output from the Debugger

You can specify formats for commands that examine locations within the program. The formats that you can specify are identical to those available to the C library routine **fprintf**. See Chapter 11, "C Library Reference," in the *C Language Manual* (SC09-1128) for a complete description of **printf** format specifiers. A summary of how to format debugger output appears in Appendix B, "Debugger Command Summary" at the back of this manual.

The debugger does not check storage alignment, so printing a *char* data object as a *short* on an invalid boundary may cause

unpredictable behavior. In the debugger, a format specification stops at the first semicolon or at the end of the input line.

Here are some examples of the debugger format specifiers and what they do:

`:x p%c`	print *x* as a *char* data object in EBCDIC
`:d p%17.5f`	print *d* as a *double* to 5 places
`:ptr p0x%08X`	print *ptr* as an *int* in hexadecimal with leading 0x
`:x p%o;:x u025`	set the format for *x* to octal and put a newline in *x*

Printing the Contents of Data Objects

The debugger can access data objects both directly and by indirection on a pointer, as in the C language itself. You specify indirection on a pointer data object by writing:

:*<*location*>

where <*location*> designates a pointer data object of the appropriate type. For example, the command

:*pi p

will print the contents of the data object pointed at by the pointer data object **pi**. The type of the data object is *char*, unless you specify otherwise.

You can specify indirection on a pointer data object only as many times as you specify the pointer attribute in your original declaration. If you request too many levels of indirection, the debugger ignores the extra requests. If you write an integer constant to designate a data object at some absolute memory address, you can specify indirection any number of times. Indirection is valid only in conjunction with location specifiers for the **p** and **u** commands; the rest of the command set ignores indirection.

You specify double indirection on the data object **z** by writing:

:z p**

Type Casts

You write type casts to make the debugger treat a data object as if it had a different type from its declaration. Unlike in C, where a type cast tells the compiler to convert the value of the data object to the specified type, the debugger type cast notation behaves as though the conversion had already taken place. Type casting, like indirection, works only with location specifiers for the **p** and **u** commands. Type casts take the form:

: (<*data_type*>**)***data_object_name* **p**

or

: **(**<*data_type*>**)***data_object_name* **u**

where acceptable types for <*data_type*> are:

[signed] [unsigned] char
[signed] [unsigned] short
[signed] [unsigned] int
[signed] [unsigned] long
float
double
long double

and *data_object_name* is a location specifier of the second kind. Specify at most two type names in a type cast. For example, to type cast **z** (a *float* in this example) to a *long*, and print the result in hexadecimal, write:

```
:(long)z p%lX
```

You can combine type casts and indirection. To print the contents of the data object pointed at by the data object **pi** as a *long*, write:

```
:*(long)pi p
```

You cannot print or update arrays directly, and you cannot subscript them to obtain a particular element.

You can print the contents of multiple adjacent memory locations, like arrays and structs, but only if the individual elements of the aggregate type are aligned on integer boundaries. Elements of type *int*, *long*, and *float* will always meet this criterion, but problems will result if elements of the structure are of type *char*, *short*, or *double*. The debugger may skip over some members as it advances to integer boundaries.

The debugger prints the warning **(no format)** when you ask it to print an aggregate data object type. This means that there is no default format associated with the data object.

Summary of Location Specifiers

There are three kinds of location specifiers:

1. For designating one or more executable source lines
2. For designating a data object
3. For providing an unsigned decimal integer.

There are three commands which allow location specifiers of the first kind: **b**, **d**, and **m**. Four commands allow location specifiers of the second kind: **b**, **d**, **p**, and **u**. Three commands allow location specifiers of the third kind: **g**, **s**, and **f**. With an **s** or **g** command, the location specifier specifies the number of source lines to process. For the **f** command it specifies the number of columns per horizontal tab stop.

A location specifier of the first kind can take the following forms:

file_name:line_number	(any executable line)
file_name:	(any executable line throughout file)
function():line_number	(current file assumed)
function():	(throughout function named)
:line_number	(current function or file assumed)
.	(the current line)

A location specifier of the second kind can take the following forms:

file_name:data_object_name
(data object with external linkage)

function():data_object_name
(in scope of function named)

:data_object_name
(in scope of current function)

number
(hexadecimal, decimal, or octal absolute address)

A data object name is an identifier currently in scope as a data object. You can change the current scope with the **m** command. You can write any number of asterisks (indicating indirection) and/or a type cast in front of a data object name.

Chapter 7: Maintaining Libraries

If you write C functions that you want to use in several programs, you can place them in an object module library. You can then use the **GLOBAL TXTLIB** command to specify that the **LOAD** command should search that library when it loads your program.

Under VM/CMS, you create object module libraries using the **TXTLIB** command. Libraries you create in this way have a filetype of **TXTLIB**.

Specifying Symbol Names

In order for the **LOAD** command to find an object module within a TXTLIB, the names of the functions and external data object that the module defines must be declared.

If your object module was compiled with the **CSECT** option, the compiler produces a named CSECT. The **TXTLIB** command will produce a member in the TXTLIB with the member name that matches the CSECT name, and an alias name for each entry point (external function) and external data object. To avoid name conflicts with library and user defined names, you must not use CSECT names which are the same as library or user function names. Also, you should avoid CSECT names beginning with '**@**', '**$**', '**#**', and '**C$**' because the C library makes use of these symbols internally.

If you do not specify a CSECT name, you must add **NAME** and **ALIAS** statements for each external function and data object. You must associate a single **NAME** statement with each object module. Associate one or more **ALIAS** statements with the module if the module contains more than one external function or data object. The statements have the format:

    (space)NAME <*NAME*>

and

    (space)ALIAS <*NAME*>

where <*NAME*> is the symbol name of the entry point. <*NAME*> will not exactly match the file name as it appears in your C source. See Chapter 8, "Interfaces with Other Languages," for information on the conversion of C names to object module symbol names.

You add the **ALIAS** statements and then the **NAME** statement to the end of the **TEXT** file. The **NAME** statement must be the last record of the **TEXT** file. You can append the statements using

XEDIT or **COPYFILE** with the **APPEND** option. For files you compile frequently, maintain a short file of **NAME** and **ALIAS** statements and append it to the **TEXT** file after each compilation. Do this using a statement of the form:

```
COPYFILE SUBA TEXTNAME A SUBA TEXT A ( APPEND
```

Creating TXTLIB Files

Use the **GEN** option of the **TXTLIB** command to create object module libraries.

The example below creates the library **MYSUBS TXTLIB**, containing the object module **SUBA TEXT**.

```
TXTLIB GEN MYSUBS SUBA
```

Adding a Member to a TXTLIB File

The **ADD** option of the **TXTLIB** command adds members to a TXTLIB. To add the program **SUBB TEXT** to the TXTLIB **MYSUBS TXTLIB**, enter:

```
TXTLIB ADD MYSUBS SUBB
```

The name you use when you add a member to a TXTLIB is the CMS file name.

Deleting a Member of a TXTLIB File

To delete members of a TXTLIB, and compress the library, use the **DEL** option. The name you use when deleting members is the symbol name you provide in the **NAME** statement for that member.

The following example deletes the member **SUBB** in **MYSUBS TXTLIB**

```
TXTLIB DEL MYSUBS SUBB
```

Replacing a TXTLIB File Member

To replace members of a TXTLIB, you must first delete members with the **DEL** command, and then add new members to the TXTLIB with the **ADD** command as described above. You use the name you provide in the **NAME** statement to name members you want to delete. You use CMS file names when naming members you want to add.

Listing TXTLIB File Members

The **MAP** option of **TXTLIB** lists members of a TXTLIB. You can also use the CMS commands that have **MEMBER** options to reference members of your TXTLIB. Read *CMS User's Guide* (SC19-6210) for more information on the **TXTLIB** command and CMS commands with **MEMBER** options.

Using Libraries

Use the **GLOBAL TXTLIB** command to make the members of a TXTLIB available to the loader. The command

```
GLOBAL TXTLIB CLIB MYLIB
```

specifies that the loader should search the library **MYLIB** in addition to the standard C library.

Refer to Chapter 4, "Loading and Generating Programs," for more information on using the **GLOBAL** command to access the C library. See the *VM/System Product CMS Command and Macro Reference* (SC19-6209) for more information on the **TXTLIB** and **GLOBAL** commands.

Chapter 8: Interfaces with Other Languages

The C compiler supports calls to IBM assembly language routines from within a C program. Once called from within a C program, an assembly language routine can access C data objects or call C functions. You can also call functions written using the standard OS calling sequence from C.

Calling Assembly Language Functions from C

You call an assembly language function from C the same way you call a C function. The assembly language function must match the C conventions for:

* Mapping C identifiers to external names
* Transferring control to another function
* Preserving registers
* Passing argument data objects
* Using the stack
* Returning a value
* Returning control to the calling function

This section addresses these issues in the order given.

Mapping C Identifiers to External Names

When you declare a function with external linkage, its external name is formed from the first eight characters of the identifier. All letters are forced to a single case, and all underscores are replaced with @ characters. For example:

function identifier	*external name*
abc	ABC
ABC	ABC
get_more	GET@MORE
hypotenuse	HYPOTENU

The entry point of your assembly language function must match the external name that C uses.

When you declare a data object with external linkage, its external name is formed from the first seven characters of the identifier. All letters are forced to a single case, and all underscores are replaced with @ characters. A $ character is prepended to each

name. For example:

data object identifier	*external name*
abc	**$ABC**
ABC	**$ABC**
get_more	**$GET@MOR**
Hypotenuse	**$HYPOTEN**

The representation of data objects in C is described in your *C Language Manual* (SC09-1128) in Chapter 4, "Declarations."

Transferring Control to Another Function

A C function calls another function by placing its address in register 15 (R15) and executing the instruction:

```
BALR 14,15
```

The address of the called function is in R15. The return address is in R14.

Your assembly language function must preserve the contents of R14 to return control to the C function that calls it. It may make use of R15 to address instructions and data within the assembly language function. If there are arguments to the function, your function must use R10 to locate them, as described below.

Preserving Registers

A C function expects that nearly all registers have the same content upon return from the called function that they have when the function is first called. The only exceptions are:

R12 – which is used to return values of integer or pointer type

R14 – which may be in any state

R15 – which may be in any state

F0 – which is used to return values of floating type

F2 – which may be in any state

F4 – which may be in any state

F6 – which may be in any state.

Your assembly language function can make use of a "save area" to preserve all registers that it might modify. The save area consists of 18 words, each 4 bytes long, on a fullword storage boundary. The calling function assures that R13 points to the save area when your assembly language function is entered. By convention, the word at 4(R13) contains the address of a previous save area. The C source level debugger assumes, for example, that all save areas are linked together in this fashion. The debugger also assumes that registers are stored on function entry with the instruction:

```
STM 14,11,12(13)
```

You must follow these conventions in your assembly language function if you want it to function properly with the C source level debugger.

Passing Argument Data Objects

A C function stores all argument data objects in an area pointed at by R10. The area begins with another 18 word save area, on a fullword storage boundary. The calling function sets aside this area entirely for the convenience of the called function. Your assembly language function need not use it. You can use this new save area if your assembly language function must call yet another function. A useful sequence of instructions at function entry is:

```
STM  14,11,12(13)
ST   13,4(10)
LR   13,10
```

which saves the registers as described above, links the new save area to the older one, and makes the new save area the current one for the function. Your assembly language function is now ready to call another function and provide it with a save area pointed at by R13.

Argument data objects are just above this new save area. If the function returns a structure type, the arguments begin at 76(R10) and continue to higher addresses. For all other functions, the arguments begin at 72(R10). All values of pointer type occupy four bytes. All values of integer type are converted by the calling function to a four byte integer representation. All values of floating type are converted by the calling function to an eight byte *double* representation, beginning at an offset that is a multiple of eight bytes. Therefore, if you call a function with a mixture of argument types, there may be four byte "holes" before some of the arguments of floating type.

All arguments of structure type begin at an offset that is a multiple of four bytes, if the structure contains no component of type *double* or *long double*. An argument of structure type begins at an offset that is a multiple of eight bytes if the structure contains any component of type *double* or *long double*. If you call a function with a mixture of argument types, therefore, there may be four byte holes before some of the arguments of structure type. There may also be one to seven byte holes *after* arguments of structure type. For example:

```
struct {
    char c1, c2;
    } cs;
int i, j;
double d;

j(cs, i, j, d);
```

will place

cs at 72(R10) occupying 2 bytes
i at 76(R10) occupying 4 bytes
j at 80(R10) occupying 4 bytes
d at 88(R10) occupying 8 bytes

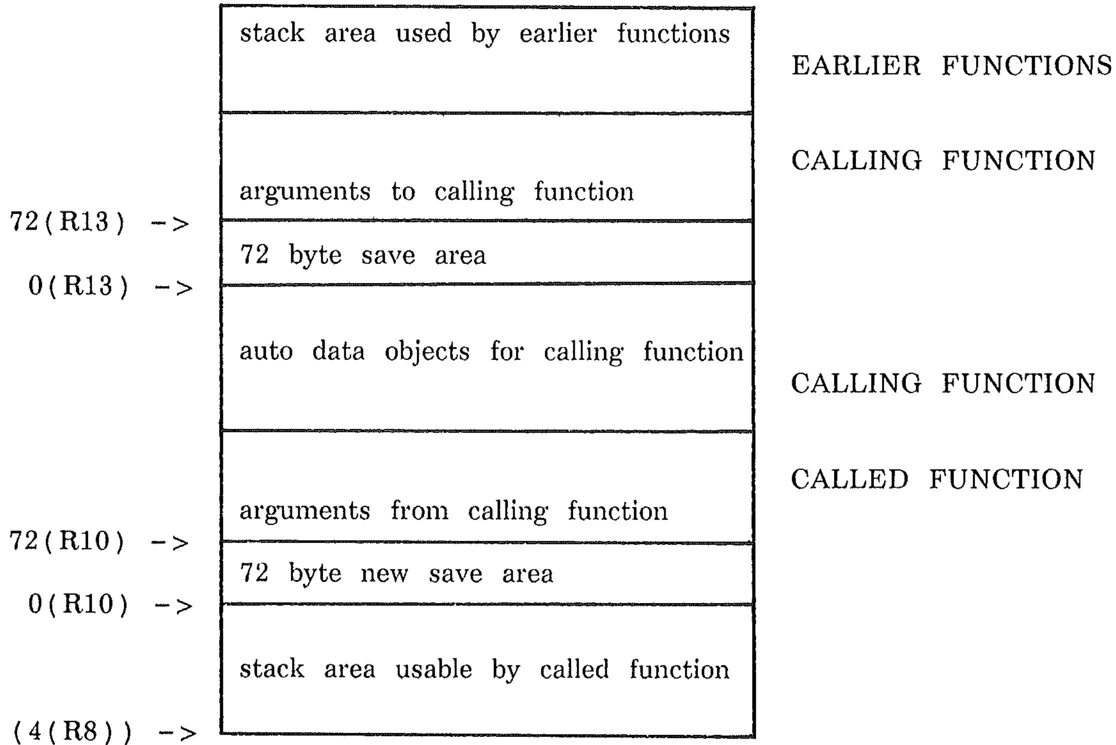
There are holes after **cs** and **j**.

Your assembly language program cannot determine how many arguments are present, or what their types are. It may alter the value stored in any of the arguments, since all are deallocated upon return.

Using the Stack

The save areas and argument data objects described above all reside within a storage area called a "stack." A C function allocates calling environments and data objects with dynamic lifetime within the stack, using a "last in first out" discipline. When your assembly language function is entered, the stack looks like:

HIGHER ADDRESSES



LOWER ADDRESSES

The area beginning at 72(R13), "arguments to calling function," is shown here to illustrate how the pattern of stack usage repeats for each function call. Your called function should never access any of these argument data objects directly, since the calling function assumes that its argument data objects are for its private use.

You can use for temporary storage the area immediately below 0(R10), as long as you do not access storage below the address

stored at 4(R8). If you call a C function from your assembly language function, however, you must allocate any storage you use in this area. You allocate storage on the stack by subtracting from R10 the number of bytes of storage you need, rounded up to a multiple of eight bytes. Remember to set aside the storage beginning at 0(R10) for the save area and argument data objects you need to call any C functions. The storage allocated for the private use of your assembly language function begins above the last argument data object for any C functions that you call.

The section "Controlling Stack Size" in Chapter 4, "Loading and Generating Programs," tells you how to alter the size of the stack area.

Returning a Value

A function returning any integer or pointer type returns with the value in R12. You must convert all integer values to a four byte representation. A function returning any floating type returns with the value in F0. You must convert all floating values to type *double*, which occupies eight bytes.

A function returning a structure type assumes that the four byte area at 72(R10) contains a pointer to a data object of the structure type. The assembly language function must store a return value in this data object before it returns. The value stored in the data object pointed at by 72(R10) is undefined when the function is first entered. There is no way to determine the size of the structure data object within your assembly language function.

A function returning *void* returns no value.

Returning Control to the Calling Function

To return from a function, compute and store the return value, restore any registers that must be preserved, and branch to the address that was in R14 when your assembly language function was first entered. If you used the entry sequence described earlier, you can return with the exit sequence:

```
L    13,4(0,13)
LM   14,11,12(13)
BR   14
```

Any code sequence that returns to the same location with the necessary registers restored is also acceptable.

Calling Functions Written in Other Languages from C

You use the OS calling sequence to call functions written in languages other than C from within a C program. You call a function that uses the OS calling sequence by declaring the function as you normally would, except that in addition you must write the preprocessor directive

```
#pragma linkage (func, OS)
```

where *func* is the name of the function you are calling. Write the preprocessor directive before any call to *func*.

Identifiers are mapped as described earlier in the chapter. Control is transferred as described earlier, by executing the instruction

```
BALR 14,15
```

with the address of the called function in R15. The called function must preserve the contents of R14 to return control to the C function that calls it. It may make use of R15 to address instructions and data within the called function. R13 points at an 18 word save area, on a fullword storage boundary, which the called function may use to store registers as described above.

If there are no arguments, R1 is set to zero by the calling function. Otherwise, R1 points to a special "argument block" which the calling function creates. To construct the argument block, the compiler processes arguments by the following rules:

* The lowest (first) entry in the argument block corresponds to the leftmost argument.
* The calling function converts each argument by the usual rules, as described in Chapter 5, "Expressions" in the section entitled "Function Calls," in your *C Language Manual* (SC09-1128).
* The compiler widens each argument to *int*, as described in Chapter 5, "Expressions" in the section entitled "Arithmetic Conversions," in your *C Language Manual* (SC09-1128). This conversion affects only arguments of an integer type represented in fewer than 32 bits.
* The calling function converts any argument of type *float* to type *double*.
* If the argument is a pointer type, the calling function adds it to the argument block unchanged.
* If the argument is not a pointer type, the calling function copies it into a temporary data object of the same type, and adds a pointer to the temporary data object to the argument block instead.
* The calling function sets the sign bit of the highest (last) entry in the argument block, which corresponds to the rightmost argument.

If you are calling a function that does not expect to be able to alter the value stored in any of its arguments, you can write

```
#pragma linkage(osfun, OS)

double osfun(double, double);
double x, y, z;

x = osfun(3.0, 2.7);
x = osfun(y, z);
```

Both function calls create temporary data objects and pass pointers to the temporary data objects in the argument block.

If you are calling an OS function that expects to be able to alter the value stored in any of its arguments and you call the function this way, the called function alters the value stored in the temporary data object, not the actual data object. To make sure that the value in the actual data object is altered, you must write:

```
#pragma linkage(osfunc, OS)

double osfunc(double *, double *);
double x, y, z;

x = osfunc(&y, &z);
```

The registers that need not be preserved by the called function are:

R0 – which is used to return values of integer type

R1 – which may be in any state

R14 – which may be in any state

R15 – which may be in any state

F0 – which is used to return values of type *double*

F2 – which may be in any state

F4 – which may be in any state

F6 – which may be in any state.

The called function again expects an 18 word save area, which R13 points at.

The called function returns a value of integer type in R0. All integer values must be converted to a 4 byte representation. The called function returns a value of floating type in F0. All floating values must be converted to an 8 byte representation, as for the C type *double*.

Most languages do not provide the ability to perform input/output and signal handling in more than one language. Generally, you perform I/O and signal handling from the language you start your program execution with.

Appendix A: Compiler Command Summary

To invoke the compiler:

```
CC <file name> [<file type>] [[<file mode>]] [(options]
```

where the file mode, file type, and option list are optional. You can specify the file type and not the file mode. If you specify the file mode you must specify the file type. The default file type is **C**. The default file mode is **A**.

Compiler options:

(NO)ASM – Produce an object module in file *<filename>* **TEXT A**. When you specify **NOASM**, the compiler writes assembler text to *<filename>* **ASSEMBLE A** and does not generate an object module. Default is **ASM**.

CSect(name) – Produce a control section (CSECT) with name *name*. When you specify **RENT** in combination with **CSECT** the compiler also produces a second CSECT with name $*name*. *name* can be up to seven characters long when you specify **CSECT** in combination with **RENT**, and eight characters otherwise. The CSECT name you specify must not conflict with external function names or external data object names.

(NO)DEBug – Produce debugger entry points to enable linking with debugger modules. **DEBUG** is invalid in combination with **RENT**. **DEBUG** disables register allocation. Default is **NODEBUG**.

Define(name=def,name=def, ...) – Define a symbol at compile time. * represents a definition of the form *name=def*. If you omit *=def*, the compiler assumes the definition to be "1". You may specify up to ten definitions.

(NO)HASM – **HASM** invokes Assembler-H. **NOHASM** invokes Assembler-XF. Default is to invoke the assembler specified at installation time as the default assembler.

(NO)IDENT8 – Restrict the number of significant internal identifiers to 8 characters. Default is **NOIDENT8** (31 characters).

LINecount(n) – Use *n* as the number of lines to be included in each page of a listing, including heading lines and blank lines. When you request a C source listing only, *n* can be any integer between 5 and 32767. When you request a

listing that contains assembly language source, *n* can be any integer between 5 and 99 if you use Assembler-H or between 5 and 999 if you use Assembler-XF. Default is 60.

(NO)LISt and **(NO)SOurce** – If you specify **LIST**, the compiler produces an assembly language listing unless it detects errors. If the compiler detects errors when you specify **LIST** it produces a C source listing interspersed with error messages. If you specify **SOURCE**, the compiler produces a C source and error listing. If you specify both **SOURCE** and **LIST**, the compiler produces an interspersed C source and assembly language listing unless it detects errors. If it detects errors when you specify both, the compiler produces a C source listing interspersed with error messages. If you specify two or more options which conflict, and are therefore invalid, the compiler generates an error message and halts. The following flags are invalid in the combinations listed:

PPONLY and **LIST**
PPONLY and **SOURCE**
LIST and **NOASM**

Default is **NOLIST** and **NOSOURCE**.

LSEarch(mod) – Define a search modifier *mod* to specify the minidisk you want the compiler to search first for **#include** files you enclose in double quotation marks. If the **#include** file is not on the *mod* minidisk the compiler searches all subsequent minidisks in order. If the compiler does not find the **#include** file it continues to search for it in the order you specify for included files enclosed in angle brackets. Default is to search for **#include** files in the order you specify for included files enclosed in angle brackets.

(NO)MACarg – Expand macro arguments in string literals as the UNIX™ System V C compiler does. Default is **NOMACARG**.

(NO)PPonly – Execute the C preprocessor to expand all macros and show all include files in the C source, but do not compile the source. The compiler writes the expanded source to **fn expanded** under CMS. Default is **NOPPONLY**.

(NO)REnt – Produce reentrant code by making every access to a data object through a base register. You must specify **CSECT** in combination with **RENT**. **DEBUG** is invalid in combination with **RENT**. Default is **NORENT**.

SEarch(mod) – Define a search modifier for **#include** files enclosed in angle brackets < >. *mod* represents the minidisk the compiler will search first. All subsequent disks are then searched in order. Default is to search all accessed minidisks if you omit the search modifier.

UNIX is a registered trademark of AT&T Bell Laboratories

SEQ(m,n)|NOSEQ – Remove line numbers from leftmost position *m* to rightmost position *n*. Default is **NOSEQ** for variable record length format files and **SEQ(73,80)** for fixed record format files. If you specify **NOSEQ** in combination with **SEQ(*m,n*)**, the compiler generates an error message and halts.

(NO)SHowinc – Show contents of include files in listing. Default is **NOSHOWINC**, which does not show the contents of included files.

(NO)TErminal – **NOTERMINAL** tells the compiler not to write error messages to your terminal screen. If you specify **NOTERMINAL**, the compiler sends error messages to the file <*filename*> **LISTING A** unless you also request a listing. Default is **TERMINAL**, which sends error messages to your terminal screen.

(NO)UPCONV – Select UP (unsignedness preserving) rules for type conversion. Default is **NOUPCONV**, which selects value preserving rules as in the proposed ANSI standard.

Appendix B: Debugger Command Summary

To enable debugging, type:

```
CC <program name> (DEBUG
```

where *<program name>* is the name of the program you want to debug.

Run program to next breakpoint	g
Run program to breakpoint *n*	*n*g
Step through program line by line	<ENTER>
Step through multiple (*n*) lines	*n* s
Exit debugging session	q
Print contents of a data object	:*data_object_name* p
Print contents using a pointer	:**pointer_name* p
Print character string using a pointer	:**pointer_name* p%s
Display current line	.
Display file name	f
Execute multiple debugger commands	*command*;*command*
Create log of debugger output	w *file_name*
Set breakpoint for a data object	*data_object_name* b [*cmd*]
Set a breakpoint for a line	*line_number* b
Delete breakpoint for a data object	*data_object_name* d
Delete a breakpoint for a line	*line_number* d
Change location by stack frames:	
up one frame	mu
down one frame	md
top of stack	mt
bottom of stack	mb
specific location	*location_specifier*m
Display list of stack frames from current frame to top of stack	t
List data object names and type for each frame	tv
List names, types, and addresses for each frame	tv&
Update value of specified location	u *new_value*
or	= *new_value*
List names and types for all data objects in current frame	v
List names, types, and addresses for all data objects in current frame	v&
Redirect command output to a file	>*file_name*
Redirect command input from a file	<*file_name*

Format Specifiers

One of the set of specifiers enclosed in braces is mandatory. Those enclosed in brackets are optional.

```
%[+|-| |#][:>nn][.:>nn][h|l] {c|d|i|o|u|x|X|e|E|f|g|G|s|p}
```

A format specifier has the following parts:

* a percent sign **%**
* an optional flag +, –, #, or space
* an optional field width, as a decimal integer constant
* an optional decimal point followed by a precision, as a decimal integer constant
* an optional size qualifier **h**, **l**, or **L**
* a conversion character from the set

  {**c d i o u x X e E f g G s p**}

Type Casts

Type casts must take one of the forms

:(*data_type*)*data_element* p

or

:(*data_type*)*data_element* u

where *data_type* must be one of: {*char, short, unsigned short, int, unsigned int, long, unsigned long, float, double, long double*} and *data_element* must be a data object.

Appendix C: EBCDIC Character Table

Hexadecimal Value	EBCDIC	Hexadecimal Value	EBCDIC	Hexadecimal Value	EBCDIC	Hexadecimal Value	EBCDIC
00	NUL	10	DLE	20	DS	30	
01	SOH	11	DC1	21	SOS	31	
02	STX	12	DC2	22	FS	32	
03	ETX	13	DC3	23	WUS	33	IR
04	SEL	14	RES/ENP	24	BYP/INP	34	PP
05	HT	15	NL	25	LF	35	TRN
06	RNL	16	BS	26	ETB	36	NBS
07	DEL	17	POC	27	ESC	37	EOT
08	GE	18	CAN	28	SA	38	SBS
09	SPS	19	EM	29	SFE	39	IT
0A	RPT	1A	UBS	2A	SM/SW	3A	RFF
0B	VT	1B	CU1	2B	CSP	3B	CU3
0C	FF	1C	IFS	2C	MFA	3C	DC4
0D	CR	1D	IGS	2D	ENQ	3D	NAK
0E	SO	1E	IRS	2E	ACK	3E	
0F	SI	1F	IUS/ITB	2F	BEL	3F	SUB
40	SP	50	&	60	–	70	
41	RSP	51		61	/	71	
42		52		62		72	
43		53		63		73	
44		54		64		74	
45		55		65		75	
46		56		66		76	
47		57		67		77	
48		58		68		78	
49		59		69		79	‘
4A	¢	5A	!	6A	|	7A	:
4B	.	5B	$	6B	,	7B	#
4C	<	5C	±	6C	%	7C	@
4D	(	5D	)	6D	_	7D	'
4E	+	5E	;	6E	>	7E	=
4F		5F	¬	6F	?	7F	"

Hexadecimal Value	EBCDIC	Hexadecimal Value	EBCDIC	Hexadecimal Value	EBCDIC	Hexadecimal Value	EBCDIC
81	a	91	j	A1	~	B1	
82	b	92	k	A2	s	B2	
83	c	93	l	A3	t	B3	
84	d	94	m	A4	u	B4	
85	e	95	n	A5	v	B5	
86	f	96	o	A6	w	B6	
87	g	97	p	A7	x	B7	
88	h	98	q	A8	y	B8	
89	i	99	r	A9	z	B9	
8A		9A		AA		BA	
8B		9B		AB		BB	
8C		9C		AC		BC	
8D		9D		AD	[	BD	]
8E		9E		AE		BE	
8F		9F		AF		BF	
C0	{	D0	}	E0	\	F0	0
C1	A	D1	J	E1	NSP	F1	1
C2	B	D2	K	E2	S	F2	2
C3	C	D3	L	E3	T	F3	3
C4	D	D4	M	E4	U	F4	4
C5	E	D5	N	E5	V	F5	5
C6	F	D6	O	E6	W	F6	6
C7	G	D7	P	E7	X	F7	7
C8	H	D8	Q	E8	Y	F8	8
C9	I	D9	R	E9	Z	F9	9
CA	SHY	DA		EA		FA	⌡
CB		DB		EB		FB	
CC	graphic	DC		EC	chair	FC	
CD		DD		ED		FD	
CE	fork	DE		EE		FE	
CF		DF		EF		FF	EO

Index

#define 3-4
#include 3-4
#include files 3-3
#include files; location 3-3
#include files; search order 3-3
#pragma linkage directive 8-5
abnormal program termination 4-3
add TXTLIB members 7-2
address of argument block 8-6
address of arguments 8-3
address specifiers; debugger 6-2
allocation of calling environments 4-3
ANSI standard 4-10
architecture; compiler 1-1
argc argument 5-1
argument block address 8-6
argument data objects; pass 8-3
arguments in macros 3-4
arguments of struct type; offset 8-3
arguments to program 5-1
arguments; command 5-1
arguments; start address 8-3
argv array 5-1
arrays; declare 4-3
Assembler-H 3-2
Assembler-XF 3-2
assembly language listings 3-3
assembly language; call from C 8-1
assembly language; calls to 8-1
automatic register allocation 3-2
base address register 4-4
base register address 3-4
breakpoint; set and delete 6-6
C calling conventions 8-1
C library 4-5
C runtime environment 4-5
C source listing 3-3
C stack 4-3
C#INIT1 control section 4-5
C$START 4-2
C$TEXT object module 4-3
C; call from assembly language 8-1
call C from assembly language 8-1
calling conventions; C 8-1
calling environment; allocation 4-3
calling function; return control to 8-5
calls to assembly language 8-1
character set 2-1
columns; ignore 3-4
command arguments 5-1
command summary; debugger 6-2
compiler architecture 1-1
compiler commands 3-1
compiler; introduction 1-1
compiler; run 3-1
control section 3-2, 4-4, 4-5
control; return to caller 8-5
control; transfer 8-2
CP nucleus 4-4, 4-6
create libraries 7-1
CSECT 3-2, 4-4
CSECT option 4-4
data object; static 4-4
data objects; pass 8-3
debugger 3-2
debugger command output; log of 6-6
debugger; address specifiers 6-10
debugger; breakpoints 6-6
debugger; command summary 6-2
debugger; demonstration 6-1
debugger; display line and file 6-5
debugger; enable 6-1
debugger; examine data objects 6-4
debugger; get function name and line 6-2
debugger; get source line contents 6-2
debugger; introduction 6-1
debugger; multiple commands 6-5
debugger; single step execution 6-3
debugging; type casts 6-9
declare arrays 4-3
declare functions with external linkage 8-1
default behavior 3-1
default record format 2-1
default record length 2-1
define identifier 3-2
delete breakpoint 6-2
delete TXTLIB members 7-2
discontiguous saved segment 4-4, 4-6, 4-7
display file; debugger 6-2
dummy sections; external 4-4
echo.c; source listing 6-1
enable debugging 6-1

enter multiple commands; debugger 6-2
enter programs 2-1
entry point 4-2
entry point; specify 4-3
entry points; specify 5-1
error listing 3-3
error messages 3-1
example source code 6-1
execute load module 5-1
execute object module 5-1
expansion of macros 3-4
external dummy sections 4-4
external identifier 4-5
external linkage; declare functions 8-1
external names; form 8-1
external names; map from C identifiers 8-1
file inclusion 3-4
files; variable length record format 3-4
floating values 8-3
form external names 8-1
function; return control to caller 8-5
get function name; debugger 6-2
GLOBAL TXTLIB 4-1
holes 8-3
identifier names; map 8-6
identifiers 3-2
identifiers; map to external names 8-1
include a file 3-4
increase stack size 4-3
indirection 6-2
Initial Program Loader 4-7
integer values 8-3
internal identifiers 3-2
introduction to compiler 1-1
length of line 2-1
levels of indirection 6-9
libraries; create 7-1
library members; reference 7-2
library; object module 4-1
library; runtime 4-1
line length 2-1
linkage editor 1-1
list TXTLIB members 7-2
listing options 3-3
listing; C source 3-3
listing; error 3-3
listings 1-2
listings; assembly language 3-3
load module; execute 5-1
load programs 4-1
log debugger command output 6-2
macro arguments 3-4
macro expansion 3-4
map C identifiers to external names 8-1
mapping identifier names 8-6
module; nonrelocatable 4-2
modules not produced by C 4-3
no format; error diagnostic 6-10
nonrelocatable module 4-2
object module 3-2
object module library 4-1
object module; execute 5-1
options for listings 3-3
options; compiler 3-1
options; request 3-1
OS calling sequence 8-1, 8-5
parameterized macros; define 3-2
pass argument data objects 8-3
pointer values 8-3
preprocessor 3-4
preprocessor directive; #pragma linkage 8-5
preserve registers 8-2
program arguments 5-1
program entry; instructions 8-3
program termination; abnormal 4-3
programs; enter 2-1
programs; load 4-1
record format; default 2-1
record length; default 2-1
reentrancy 3-2
reentrant code 3-4
reentrant program 4-4, 4-5, 4-6
reference library members 7-2
reference TXTLIB members 7-2
register allocation; automatic 3-2
registers; preserve 8-2
RENT option 4-4
replace TXTLIB members 7-2
request options 3-1
restrictions on source file 2-1
return a value 8-5
return control to caller 8-5
RLINK program 4-4
running the compiler 3-1
runtime environment 4-5
runtime library 4-1
save area 8-2
search modifier 3-3

search order 3-1, 3-3, 3-4
set breakpoint 6-2
signal SIGSTACK 4-3
SIGSTACK signal 4-3
size of stack 4-3
source code; example program 6-1
source columns; ignore 3-4
source file; file type 2-1
source file; restrictions 2-1
source level debugger 6-1
source line contents; debugger 6-2
specify entry points 5-1
stack 4-3
stack area 8-4
stack frames 6-7
stack overflow 4-3
stack size 4-3
stack size; increase 4-3
standard error stream 5-1
standard input stream 5-1
standard output stream 5-1
starting storage location 4-3
static data object 4-4
stderr text stream 5-1
stdin text stream 5-1
stdout text stream 5-1
storage keys 4-7
storage location; specify 4-3
struct arguments; offset 8-3
termination; abnormal 4-3
transfer control to function 8-2
trigraphs 2-1
TXTLIB member; add 7-2
TXTLIB member; delete 7-2
TXTLIB member; list 7-2
TXTLIB members; reference 7-2
TXTLIB members; replace 7-2
type casts in debugging 6-9
undefined symbols 4-2
unsignedness preserving 3-5
value preserving 3-5
values; return 8-5
variable length record format files 3-4
virtual machine 4-6
VM/CMS 4-5
XEDIT 2-1

C Compiler User's Guide
For VM/CMS

SC09-1130-01

READER'S COMMENT FORM

Please use this form only to identify publication errors or to request changes in publications. Direct any requests for additional publications, technical questions about IBM systems, changes in IBM programming support, and so on, to your IBM representative or to your nearest IBM branch office. You may use this form to communicate your comments about this publication, its organization, or subject matter **with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.**

☐ If your comment does not need a reply (for example, pointing out a typing error), check this box and do not include your name and address below If your comment is applicable, we will include it in the next revision of the manual

☐ If you would like a reply, check this box Be sure to print your name and address below

Page number(s) Comment(s):

Please contact your nearest IBM branch office to request additional publications

Name ______________________________

Company or
Organization ______________________________

Address ______________________________

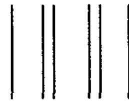
____

SC09-1130-01

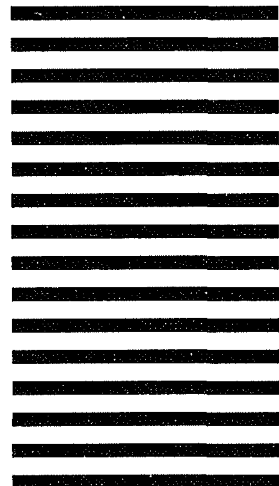
Reader's Comment Form

Cut or Fold Along Line

Fold and tape — **Please Do Not Staple** — Fold and tape



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



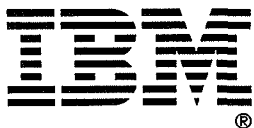
BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO 40 ARMONK, N Y

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 6R1T
180 Kost Road
Mechanicsburg, Pennsylvania 17055

Fold and tape — **Please Do Not Staple** — Fold and tape



READER'S COMMENT FORM

Please use this form only to identify publication errors or to request changes in publications Direct any requests for additional publications, technical questions about IBM systems, changes in IBM programming support, and so on, to your IBM representative or to your nearest IBM branch office You may use this form to communicate your comments about this publication, its organization, or subject matter **with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.**

☐ If your comment does not need a reply (for example, pointing out a typing error), check this box and do not include your name and address below. If your comment is applicable, we will include it in the next revision of the manual.

☐ If you would like a reply, check this box Be sure to print your name and address below

Page number(s): Comment(s):

Please contact your nearest IBM branch office to request additional publications.

Name ______________________________

Company or
Organization ______________________________

Address ______________________________

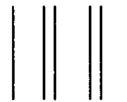
____

SC09-1130-01

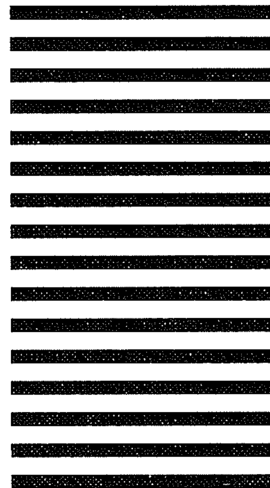
Reader's Comment Form

Cut or Fold Along Line

Fold and tape | Please Do Not Staple | Fold and tape



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 6R1T
180 Kost Road
Mechanicsburg, Pennsylvania 17055

Fold and tape | Please Do Not Staple | Fold and tape

READER'S COMMENT FORM

Please use this form only to identify publication errors or to request changes in publications Direct any requests for additional publications, technical questions about IBM systems, changes in IBM programming support, and so on, to your IBM representative or to your nearest IBM branch office You may use this form to communicate your comments about this publication, its organization, or subject matter **with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.**

☐ If your comment does not need a reply (for example, pointing out a typing error), check this box and do not include your name and address below If your comment is applicable, we will include it in the next revision of the manual.

☐ If you would like a reply, check this box. Be sure to print your name and address below.

Page number(s)· Comment(s):

Please contact your nearest IBM branch office to request additional publications.

Name ______________________________

Company or
Organization ______________________________

Address ______________________________

____

SC09-1130-01

Reader's Comment Form

Cut or Fold Along Line

Fold and tape | Please Do Not Staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

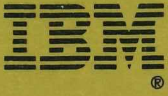
BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 6R1T
180 Kost Road
Mechanicsburg, Pennsylvania 17055

Fold and tape | Please Do Not Staple | Fold and tape



International Business Machines Corporation

Order Number SC09-1130-01

File Number S370-99

Printed in U.S.A.