MICROPROCESSOR APPLICATIONS
REFERENCE BOOK

VOLUME 1

July 1981
Zilog's name has become synonymous with logic innovation and advanced microprocessor architecture since the introduction of the Z80™ CPU in 1975. The Zilog Family of microprocessors and microcomputers has grown to include the products listed in the table below. Each product exhibits special features that make it stand above similar products in the semiconductor marketplace. These special features have proven to be of substantial aid in the solution of microprocessor design problems.

This reference book contains a collection of application information about Zilog microprocessor products. It includes technical articles, application notes, concept papers, and benchmarks. The reference book is intended as the first of several such volumes. We at Zilog believe that designing innovative microprocessor integrated circuit products is only half the key that unlocks the future of microprocessor-based end products: the other half is the creative application of those products. Advanced microprocessor products and their creative application lead to end product designs with more features, more simply implemented, at a lower system cost. It is hoped this reference book will stimulate new product design ideas as well as fresh approaches to the design of traditional microprocessor-based products.

The material in this book is believed to be accurate and up-to-date. If you do find errors, or would like to offer suggestions for future application notes, we would appreciate hearing from you. Correction inputs should be directed to Components Division Technical Publications, and application suggestions should be directed to Components Division Application Engineering.

### Z8000 FAMILY

<table>
<thead>
<tr>
<th>Z8601</th>
<th>8-bit Single Chip Microcomputer, 2K Bytes ROM and 144 Bytes RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z8601</td>
<td>Mask Programmed</td>
</tr>
<tr>
<td>Z8602</td>
<td>Development Package</td>
</tr>
<tr>
<td>Z8603</td>
<td>Prototopack</td>
</tr>
<tr>
<td>Z8671</td>
<td>Basic/Debug</td>
</tr>
<tr>
<td>Z8681</td>
<td>ROMless</td>
</tr>
<tr>
<td>Z8000 FAMILY</td>
<td>16-bit General Purpose Microprocessor</td>
</tr>
<tr>
<td>Z8001</td>
<td>CPU</td>
</tr>
<tr>
<td>Z8002</td>
<td>CPU</td>
</tr>
<tr>
<td>Z8003</td>
<td>VMPP</td>
</tr>
<tr>
<td>Z8010</td>
<td>MMU</td>
</tr>
<tr>
<td>Z8015</td>
<td>MMU</td>
</tr>
<tr>
<td>Z8030</td>
<td>Z-SCC</td>
</tr>
<tr>
<td>Z8036</td>
<td>Z-SIo</td>
</tr>
<tr>
<td>Z8038</td>
<td>Z-FIO</td>
</tr>
<tr>
<td>Z8052</td>
<td>Z-CRTC</td>
</tr>
<tr>
<td>Z8060</td>
<td>Z-FIFO</td>
</tr>
<tr>
<td>Z8065</td>
<td>Z-BEP</td>
</tr>
<tr>
<td>Z8068</td>
<td>Z-DCP</td>
</tr>
<tr>
<td>Z8090</td>
<td>Z-UPC</td>
</tr>
<tr>
<td>Z8500 FAMILY</td>
<td>Universal Peripherals</td>
</tr>
<tr>
<td>Z8530</td>
<td>SCC</td>
</tr>
<tr>
<td>Z8536</td>
<td>CIO</td>
</tr>
<tr>
<td>Z8590</td>
<td>UPC</td>
</tr>
</tbody>
</table>

### Z8600 FAMILY

<table>
<thead>
<tr>
<th>Z8610</th>
<th>8-bit Single Chip Microcomputer, 4K Bytes ROM and 144 Bytes RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z8610</td>
<td>Mask Programmed</td>
</tr>
<tr>
<td>Z8612</td>
<td>Development Package</td>
</tr>
<tr>
<td>Z8613</td>
<td>Prototopack</td>
</tr>
<tr>
<td>Z80 FAMILY</td>
<td>8-bit General Purpose Microprocessor</td>
</tr>
<tr>
<td>Z8400</td>
<td>CPU</td>
</tr>
<tr>
<td>Z8410</td>
<td>DMA</td>
</tr>
<tr>
<td>Z8420</td>
<td>P10</td>
</tr>
<tr>
<td>Z8430</td>
<td>CTC</td>
</tr>
<tr>
<td>Z8440</td>
<td>S10</td>
</tr>
<tr>
<td>Z8449</td>
<td>S10/9</td>
</tr>
<tr>
<td>Z8470</td>
<td>DART</td>
</tr>
</tbody>
</table>

### Z8500 FAMILY

<table>
<thead>
<tr>
<th>Z6000 FAMILY</th>
<th>Memory Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z6132</td>
<td>Quasi-Static RAM</td>
</tr>
</tbody>
</table>
## Table of Contents

### Z8 Single-Chip Microcomputer Family

- Advanced Architecture of the Z8 Microcomputer ........................................ 1-3
- A Comparison of MCU Units ........................................................................ 1-15
- Z8600 Interrupt Request Register ................................................................. 1-29
- 12-Bit Addressing with the Z8600 Family ....................................................... 1-31
- Z8 Family Software Framing Error Detection ................................................ 1-33
- A Programmer's Guide to the Z8 Microcomputer ............................................. 1-35

### Z80 8-Bit Microprocessor Family

- Get Powerful Microprocessor Performance .................................................... 2-2
- Designing a Microprocessor Driven Multipurpose Peripheral Controller ......... 2-13
- Z80 Family Interrupt Structure ..................................................................... 2-23
- A Z80-Based System Using the DMA with SIO ............................................. 2-23
- Using the Z80 SIO in Asynchronous Communications .................................... 2-37
- Using the Z80 with SDLC .............................................................................. 2-71
- Binary Synchronous Communication Using the Z80 SIO .................................. 2-84
- Serial Communication with the Z80A DART .................................................. 2-87
- Interfacing 8500 Peripherals to the Z80 ......................................................... 2-90
- Serial Clock Generation Using the Z8536 C10 ............................................... 2-103
- Timing in an Interrupt-Based System with the Z80 CTC ............................... 2-107
- Interfacing 16-Pin Dynamic RAMs to the Z80A Microprocessor .................... 2-119
- Designing a Microprocessor Driven Multipurpose Peripheral Controller ......... 2-147

### Z8000 16-Bit Microprocessor

- Advanced Architectural Features of the Z8000 CPU ...................................... 3-3
- A Small Z8000 System .................................................................................. 3-17
- An Introduction to the Z8010 MMU Memory Management Unit ..................... 3-28
- An Introduction to Memory Management ...................................................... 3-49
- The Z8000 and the 68000 Concept Papers: Introduction .............................. 3-57
- Z8000 vs 68000 Register Architecture .......................................................... 3-58
- Memory-Mapped vs Explicit I/O .................................................................... 3-61
- Address/Data Bus ............................................................................................ 3-62
- Segmented vs Linear Addressing .................................................................... 3-63
- Operating System Support Features ............................................................... 3-74
- A Tale of Four Microprocessors: Benchmarks Quantify Performance ............ 3-84
- Architectural Concepts for Microprocessor Peripheral Families ..................... 3-103

### Z-Family Memories

- Interfacing the Z6132 Intelligent Memory ..................................................... 4-3

### Z-BUS

- Z-BUS Component Interconnect ..................................................................... 5-3
- Z-BUS and Peripheral Support Packages Tie Distributed Computer Systems Together 5-19

### General

- An Optimizing Driver for NEC Spinwriters and Diablo Printers ...................... 6-3
- ZRTS 8000 Zilog Real-Time Software for the Z8000 Microprocessor .............. 6-61
- Peripheral Controller Chip Ties into 8- and 16-Bit Systems ............................ 6-65
- Adapting Unix to a 16-Bit Microcomputer ..................................................... 6-71
- Major Firms Join Unix Parade ....................................................................... 6-77
INTRODUCTION

The semiconductor industry accomplished dramatic technological advances in the area of MOS integrated circuit microprocessors during the 1970's, and as the next decade begins two trends are very clear. The first is the continued increased capability of the high-end general purpose microprocessors. Sixteen bit microprocessors will mature with additional "big machine" features, and 32-bit microprocessors will develop.

The second trend is in the area of single chip microcomputers. Single chip microcomputers are offering substantially greater processing power than when they were first introduced. Microcomputers are no longer limited to low end applications where unit cost and power dissipation are the primary design considerations.

Zilog is applying classical computer architecture concepts to the design of its microcomputer products. Upon close examination of the Zilog Z8 Microcomputer, one notices features that once were available only on general purpose bus oriented microprocessor products such as:

- separate program and data space
- the stack pointer and the PUSH and POP instructions
- 126K byte total memory address space
- vectored interrupts
- the CALL and RET (Return) instructions for procedure calls.

The trend in high-end single chip microcomputer architecture is clear and the consequences are obvious. The multi-chip solutions of today that employ 8-bit general purpose microprocessors will be replaced by more powerful 8-bit or 16-bit single chip microcomputers in the future.

This paper will discuss the architectural features of the Z8 Microcomputer and describe an application of the Z8 that takes advantage of the off chip expansion capability.

ARCHITECTURAL OVERVIEW

The architecture of the Z8 microcomputer offers many advanced processing features not previously available with single chip microcom-puters. The Z8 combines a powerful instruction set, simplified system expansion off chip, and flexible serial and parallel I/O capabilities to provide design solutions for a wide range of application problems.

The Z8 has a 16-bit Program Counter and a separate 16-bit Stack Pointer. The memory space may be extended beyond the 2K bytes of ROM and 124 bytes of RAM on chip, up to 126K bytes of program and data memory. There are 32 bits of I/O which can be configured into a variety of bit, nibble, and byte organizations, and the serial I/O port is a complete full duplex asynchronous receiver/transmitter. The Z8 interrupt structure allows the user to mask and prioritize the interrupt functions under program control, and the interrupts are directed to the appropriate service routine through 16-bit vectors in the first 12 locations of program memory. Two counter/timers are provided to off load time base generation and interval detection tasks from the Z8. The Z8 will operate with an 8 MHz clock and the exact frequency up to 8 MHz may be set with an external crystal, an external RC, or an external clock source. The Z8 operates from a single 5 volt power supply and offers a power down mode that allows the 124 general purpose registers on chip to operate from a back up battery. A Block Diagram of the Z8 is given in Figure 1.

![Z8 Block Diagram](image-url)
MEMORY SPACE AND REGISTER ORGANIZATION

Memory Space

The Z8 can address up to 126K bytes of program and data memory separately from the on-chip registers. The 16-bit program counter provides for 64K bytes of program memory, the first 2K bytes of which are internal to the Z8. The remaining 62K bytes of program memory are located externally and can be implemented with ROM, EPROM, or RAM.

The 62K bytes of data memory are also located external to the Z8 and begin with location 2048. The two address spaces, program memory and data memory, are individually selected by the Data Memory Select output (DX) which is available from Port 3.

The Program Memory Map and the Data Memory Map are shown in Figure 2.

![Figure 2 Program Memory Map And Data Memory Map](image)

External memory access is accomplished by the Z8 through its I/O Ports. When less than 256 bytes of external memory are required, Port 1 is programmed for the multiplexed address/data mode (AD0-AD7). In this configuration, 8-bits of address and 8-bits of data are time multiplexed on the 8 I/O lines for memory transfers. The memory "handshake" control lines are provided by the Address Strobe (AS), Data Strobe (DS), and the Read/Write (R/W) pins on the Z8. If program and data are included in the external memory space, the Data Memory Select (DX) function may be programmed into the Port 3 Mode register.

When this is done, the DX signal is available on line 4 of the Port 3 (P34) to select between program and data memory for external memory operations.

Port 0 is used to provide the additional address bits for external memory beyond the first 256 locations up to a full 16-bits of external memory address. It becomes immediately obvious that the first 8-bits of external memory address from Port 1 must be latched externally to the Z8 so that program or data may be transferred over the same 8 lines during the external memory transaction machine cycle. The AS, DS, and R/W control lines simplify the required interface logic. The timing for external memory transactions is given in Figure 3.

Registers

The Z8 has 144 8-bit registers including four Port registers (RO-R3), 124 general purpose registers (R4-R127), and 16 control and status register (R240-R255). The 144 registers are all located in the same 8-bit address space to allow any 28 instruction to operate on them. The 124 general purpose registers can function as accumulators, address pointers, or index registers. The registers are read when they are referenced as source registers, and written when they are referenced as destination registers. Registers may be addressed directly with an 8-bit address, or indirectly through another register with an 8-bit address, or with a 4-bit address and Register Pointer.

The entire Z8 register space may be divided into 16 contiguous Working Register Areas, each having 16 registers. A control register, called the Register Pointer, may be loaded with the most significant nibble of a Working Register Area address. The Register Pointer provides for the selection of the Working Register Area, and allows registers within that area to be selected with a 4-bit address.

The Z8 register organization is shown in Figure 4.

Stacks

The Z8 provides for stack operations through the use of a stack pointer, and the stack may be located in the internal register space or in the external data memory space. The "stack selection" bit (D2) in the Port 0-1 Mode control register selects an internal or external stack. When the stack is located internally, register 255 contains an 8-bit stack pointer and register 254 is available as a general purpose register. If an external stack is used, register 255 or registers 254 and 255 may be used as the stack pointer depending on the anticipated "depth" of the stack. When registers 254 and 255 are both used, the stack pointer is a full 16-bits wide. The CALL, RET, RETS, PUSH, and
Memory Read Cycle

Memory Write Cycle

External Memory Transaction Cycle

Figure 3

Figure 4 Register File Organization
POP instructions are Z8 instructions which include implicit stack operations.

**I/O STRUCTURE**

**Parallel I/O**

The Z8 microcomputer has 32 lines of I/O arranged as four 8-bit ports. All of the I/O ports are TTL compatible and are configurable as input, output, input/output, or address/data. The handshake control lines for Ports 0, 1, and 2 are bits from Port 3 that have been programmed through a Mode control register, except for AS, DS, and R/W which are available as separate 28 pins. The I/O ports are accessed as separate internal registers by the Z8. Ports 0 and 1 share one Mode control register, and Ports 2 and 3 each have a Mode control register for configuring the port.

Port 0 can be programmed to be an I/O port or as an address output port. More specifically Port 0 can be configured to be an 8-bit I/O port, or a 4-bit address output port (A8-A11) for external memory and one 4-bit I/O port, or an 8-bit address output port (A8-A15) for external memory.

Port 1 can be programmed as an I/O port (with or without handshake), or an address/data port (AD0-AD7) for interfacing with external memory. If Port 1 is programmed to be an address/data port, it cannot be accessed as a register.

Port 2 can be configured as individual input or output bits, and Port 3 can be programmed to be parallel I/O bits, and/or serial I/O bits, and/or handshake control lines for the other ports. Figure 5 shows the port Mode registers.

The off chip expansion capability using Ports 0 and 1 offers the added feature of being Z-Bus compatible. All Z-Bus compatible peripheral chips that are available now, and will be available in the future, will interface directly with the Z8 multiplexed address/data bus.

**Serial I/O**

As mentioned in the last section, Port 3 can be programmed to be a serial I/O port with bits 0 and 7, the serial input and serial output lines respectively. The serial I/O capability provides for full duplex asynchronous serial data at rates up to 62.5K bits per second. The transmitted format is one start bit, eight data bits including odd parity (if parity is enabled), and two stop bits. The received data format is one start bit, eight data bits and at least one stop bit. If parity is enabled, the eighth data bit received (bit 7) is replaced by a parity error flag which indicates a parity error if it is set to a ONE.

Timer/Counter T0, is the baud rate generator and runs at 16 times the serial data bit rate. The receiver is double buffered and an internal interrupt (IRQ3) is generated when a character is loaded into the receive buffer register. A different internal interrupt (IRQ4) is generated when a character is transmitted.

**COUNTER/TIMERS**

The Z8 has two 8-bit programmable counter/timers, each of which is driven by a programmable 6-bit prescaler. The T0 prescaler can be driven by internal or external clock sources, and the T0 prescaler is driven by the internal clock only. The two prescalers and the two counters are loaded through four control registers (see Figure 4) and when a counter/timer reaches the "end of count" a timer interrupt is generated (IRQ4 for T0, and IRQ5 for T1). The counter/timers can be programmed to stop upon reaching the end of count, or to reload and continue counting. Since either counter (one at a time) can have its output available external to the Z8, and Counter/Timer T1 can have an external input, the two counters can be cascaded.

Port 3 can be programmed to provide timer outputs for external time base generation or trigger pulses.

**INTERRUPT STRUCTURE**

The Z8 provides for six interrupts from eight different sources including four Port 3 lines (P30-P33), serial in, serial out, and two counter/timers. These interrupts can be masked and prioritized using the Interrupt Mask Register (register 251) and the Interrupt Priority Register (register 249). All interrupts can be disabled with the master interrupt enable bit in the Interrupt Mask Register.

Each of the six interrupts has a 16-bit interrupt vector that points to its interrupt service routine. These six 2-byte vectors are placed in the first twelve locations in the program memory space (see Figure 2).

When simultaneous interrupts occur for enabled interrupt sources, the Interrupt Priority Register determines which interrupt is serviced first. The priority is programmable in a way that is described by Figure 6.

When an interrupt is recognized by the Z8, all other interrupts are disabled, the program counter and program control flags are saved, and the program counter is loaded with the corresponding interrupt vector. Interrupts must be re-enabled by the user upon entering the service.
PORTS 0 AND 1 MODES (P01M)

PORT 2 MODE (P2M)

PORT 3 MODE (P3M)

Figure 5 Port Mode Registers

R249 INTERRUPT PRIORITY REGISTER (IPR)

Figure 6
routine (for nested interrupts), or upon returning from the interrupt service routine using the IRET instruction. The interrupt cycle process is shown in Figure 7.

There are eight instruction functional groups:

- Load
- Arithmetic
- Logical
- Program Control
- Bit Manipulation
- Block Transfer
- Rotate and Shift
- CPU Control

A summary of the Z8 instructions by function is given in Appendix A.

The Z8 addressing modes are optimized for using the internal ROM and RAM memories. Two of the reasons why this was done were; to improve code density (fewer bytes per instruction), and to reduce execution time.

**THE Z8 FAMILY**

The Z8 family emerged with three versions of the basic microcomputer; the 40-pin ROM version, the 40-pin EPROM version, and the 64-pin version. The 40-pin EPROM version is offered in a Zilog proprietary package called a ProtoPak (see Figure 8), that has a socket for an EPROM mounted permanently on top of a 40-pin DIP. This device will plug directly into the socket of a product designed for the ROM version, or can be the initial production component for a product that may ultimately be converted to the ROM version of the Z8. The 64-pin version has no internal ROM and comes in a 64-pin leadless chip carrier (see figure 9).

The eleven ROM address lines and eight ROM data lines are brought to pins on this version of the Z8. A 4K ROM version of the Z8 is planned for release toward the end of 1980.
Typeset Innovations, Inc. is a company based in Austin, Texas, that has designed a graphics computing system based on the Z8 microcomputer. The ProGrafix is a specialized electronic computational aid for use by graphics arts professionals in the sizing and pricing of graphic elements. Graphics arts professionals are exemplified by typesetting job estimators, typographers, graphic designers, printers, advertising layout artists, and book and magazine designers.

Graphic artists have in the past performed copyfitting through trial and error. With the increasing costs of graphics materials, the trial and error method has become a noticeable expense that can be minimized with a more accurate copyfitting technique.

The ProGrafix performs the following functions;

- Entry and display of values in the units of measurement which are commonly used in the typographical arts, including:
  - picas
  - points
  - picas and points
  - inches
  - ciceros
  - didots
  - ciceros and didots
  - centimeters
  - relative units

- Instantaneous, one keystroke conversion of values between any of the above units of measurement.

- Arithmetic operations using values which are expressed in any of the above units of measurement.

- One keystroke execution of an extremely accurate copyfitting algorithm which finds any unknown copyfitting value after the five known ones have been entered, from among the following copy descriptors:
  - width
  - depth
  - size
  - leading
  - type style density
  - character count

- Graphic proportional enlargement/reduction computations by finding any one of the following values after the other two are entered:
  - original size
  - reproduction size
  - enlargement/reduction ratio

- Memory storage and recall of intermediate results, pricing constants, and other user-created values.

When final packaging is complete the ProGrafix will appear similar to the drawing shown in Figure 10. The computational aid will have an 8 digit display, 7 annunciator LED's to indicate measurement units, an on/off switch, and a 40 key keyboard matrix. The key functions are defined in Table 1.

When Typeset Innovations began the design of the ProGrafix early in 1980, they looked for a microcomputer that had the following characteristics;

- A real (available) microcomputer powerful enough to do the job
- Compact coding
- Fast
- Easy to program
- External expansion of memory and I/O
The Z8 offered all of these characteristics and more. By the second quarter of 1980, the ProGrafix prototypes were working.

The prototype implementation is shown in Figure 11. External ROM and RAM were added using Port 1 and half of Port 0 (A8-A11). The ability to add more than 2K bytes of external memory with only 12 address lines (A0-A11) is possible because the Data Strobe (DS) line is only active when locations above the first 2K bytes are accessed. Memory locations from 0 to 2K bytes are internal to the Z8; locations from 2K bytes to 4K bytes (ROM) are external to the Z8 and selected by address line A11=1 and DS; and locations from 4K bytes to 6K bytes (RAM) are external to the Z8 and selected by address line A11 = 0 and DS active.

The remaining four bits of Port 0 were used to drive the Unit of Measure LED's and the "sign" for the numeric display.

Four of the I/O lines available from Port 3 were used to select one of eight digits on the numeric display through a 4 to 16 decoder and to scan the rows of the keypad. The other four I/O lines were used to read back the columns from the keypad.

One line from Port 2 was used for the fifth column input to the Z8 from the 40 key keypad. The remaining 7 I/O lines available from Port 2 were used for segment select on the numeric display.

The numeric display is "scan refreshed" by the Z8 at a rate that is approximately 100 times per second. As the digits of the display are being refreshed the keypad is scanned as a matrix of 8 by 5 keys. The counter/timers on the Z8 are both used; one to time the display refresh, and the other as a timer for keypad debounce. An external stack is used for temporary variable storage and during the servicing of interrupts. Only two Z8 interrupts are used by the ProGrafix, one for the display refresh counter and the other for the keyboard debounce timer.

The development of the software for the ProGrafix, which included a BCD Floating Point package, was done on a Zilog development system with the Z8 PLZ/ASM assembler. The object code was down loaded to a Z8 Development Module (DM) where the hardware was initially debugged. The external memory was added to the Z8DM in the space provided for wire wrap. When the system was 90% - 95% debugged, a prototype circuit board was built and the Z8 in a Protopak package with an EPROM was used for final system debug.

The production version of the ProGrafix will use an LCD numeric display instead of the LED display. This will make additional address lines available for expanding off chip memory. In addition, a printer option is planned that will connect to the serial port of the Z8.

The ProGrafix is expected to sell for under $500 without a printer and under $750 with a printer. The availability of the ProGrafix has been targeted for May of 1981.

The configuration of the ProGrafix computational aid around the Z8 provided a very flexible and powerful microcomputer system that can be expanded to accommodate a wide variety of applications by simply changing the software. Typeset Innovations is currently looking for other products that can be implemented with the hardware that was developed for the ProGrafix.

CONCLUSION

The Z8 represents the coming of age of the more powerful microcomputers. While the Z8 can be a cost effective design solution for low end applications, it can also be expanded to attack much more sophisticated design problems. The architecture of the Z8 was designed in a forward looking manner, and the integration of more capability onto the same chip is now limited only by the constraints of the integrated circuit technology.
<table>
<thead>
<tr>
<th>Key</th>
<th>Designation</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>C</td>
<td>Clears display register X and the operand register Y.</td>
</tr>
<tr>
<td>11</td>
<td>CE</td>
<td>Clears only the last value entered into display register X.</td>
</tr>
<tr>
<td>12</td>
<td>RECALL</td>
<td>Enters the contents of a designated memory register 0 through 9 into register X and into the display.</td>
</tr>
<tr>
<td>13</td>
<td>STORE</td>
<td>Stores display register X into memory registers 0 through 9, and into the parameter registers.</td>
</tr>
<tr>
<td>14</td>
<td>ORIG</td>
<td>Used to store or recall the value of the original size parameter in the proportional sizing algorithm.</td>
</tr>
<tr>
<td>15</td>
<td>WIDTH</td>
<td>Used to store or recall the value of the width (line length) parameter of the copyfitting algorithm.</td>
</tr>
<tr>
<td>16</td>
<td>PICA</td>
<td>Provides for the function of entering information in picas and points and for converting information on the display into either picas and points or decimal picas.</td>
</tr>
<tr>
<td>17</td>
<td>CICERO</td>
<td>Provides for the function of entering information in ciceros and for converting information on the display into either ciceros and didots or decimal ciceros.</td>
</tr>
<tr>
<td>18</td>
<td>REL-UN</td>
<td>Provides for the function of entering information in relative units and for converting information on the display into relative units.</td>
</tr>
<tr>
<td>19</td>
<td>REPRO</td>
<td>Used to store or recall the value of the reproduction size parameter in the proportional sizing algorithm.</td>
</tr>
<tr>
<td>20</td>
<td>DEPTH</td>
<td>Used to store or recall the value of the depth (vertical measure) parameter of the copyfitting algorithm.</td>
</tr>
<tr>
<td>21</td>
<td>POINT</td>
<td>Provides for the function of entering information in points and for converting the information on the display into points.</td>
</tr>
<tr>
<td>22</td>
<td>DIDOT</td>
<td>Provides for the function of entering information in didot points and for converting the information on the display into didot points.</td>
</tr>
<tr>
<td>23</td>
<td>RU/EM</td>
<td>Used to store or recall of the relative units per em space parameter.</td>
</tr>
<tr>
<td>24</td>
<td>RATIO</td>
<td>Used to store or recall the value of the ratio parameter in the proportional sizing algorithm.</td>
</tr>
<tr>
<td>25</td>
<td>SIZE</td>
<td>Used to store or recall the value of the type size parameter for the copyfitting algorithm and for the relative units conversion algorithm.</td>
</tr>
<tr>
<td>26</td>
<td>INCH</td>
<td>Provides for the function of entering information in inches and for converting the information on the display into inches.</td>
</tr>
<tr>
<td>27</td>
<td>CM</td>
<td>Provides for the dual function of entering information in centimeters, and for converting the information on the display into centimeters.</td>
</tr>
<tr>
<td>28</td>
<td>Double arrow</td>
<td>Interchanges the contents of display register X and operand register Y.</td>
</tr>
<tr>
<td>29</td>
<td>/</td>
<td>Divides operand register Y by display register X.</td>
</tr>
<tr>
<td>30</td>
<td>LEAD</td>
<td>Used to store or recall the value of the leading (line spacing) parameter of the copyfitting algorithm.</td>
</tr>
<tr>
<td>31</td>
<td>X</td>
<td>Multiplies display register X by operand register Y.</td>
</tr>
<tr>
<td>32</td>
<td>DENSITY</td>
<td>Used to store or recall the value of the type style density parameter of the copyfitting algorithm.</td>
</tr>
<tr>
<td>33</td>
<td>+</td>
<td>Adds display register X to operand register Y.</td>
</tr>
<tr>
<td>34</td>
<td>CHAR</td>
<td>Used to store or recall the value of the character count parameter of the copyfitting algorithm.</td>
</tr>
<tr>
<td>35</td>
<td>-</td>
<td>Subtracts display register X from operand register Y.</td>
</tr>
<tr>
<td>36</td>
<td>FIND</td>
<td>Invokes the calculation of an unknown copyfitting parameter given five known copyfitting parameters. This key is also used to solve for an unknown proportional sizing parameter given two known proportional sizing parameters.</td>
</tr>
<tr>
<td>37</td>
<td>.</td>
<td>Used to enter the decimal point of a floating-point number.</td>
</tr>
<tr>
<td>38</td>
<td>+/-</td>
<td>Reverses the sign of the value in display register X.</td>
</tr>
<tr>
<td>39</td>
<td>=</td>
<td>Invokes the last entered arithmetic operation using the X and Y registers as operands and places the result in display register X.</td>
</tr>
<tr>
<td>40</td>
<td>ON/OFF</td>
<td>Powers the microcomputer system on and off.</td>
</tr>
</tbody>
</table>
PROGRAFIX BLOCK DIAGRAM

Figure 11
### Load Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand(s)</th>
<th>Name of Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR</td>
<td>dst</td>
<td>Clear</td>
</tr>
<tr>
<td>LD</td>
<td>dst, src</td>
<td>Load</td>
</tr>
<tr>
<td>LDC</td>
<td>dst, src</td>
<td>Load Constant Data</td>
</tr>
<tr>
<td>LDE</td>
<td>dst, src</td>
<td>Load External Data</td>
</tr>
<tr>
<td>POP</td>
<td>dst</td>
<td>Pop</td>
</tr>
<tr>
<td>PUSH</td>
<td>src</td>
<td>Push</td>
</tr>
</tbody>
</table>

### Arithmetic Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand(s)</th>
<th>Name of Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>dst, src</td>
<td>Add With Carry</td>
</tr>
<tr>
<td>ADD</td>
<td>dst, src</td>
<td>Add</td>
</tr>
<tr>
<td>CP</td>
<td>dst, src</td>
<td>Compare</td>
</tr>
<tr>
<td>DA</td>
<td>dst</td>
<td>Decimal Adjust</td>
</tr>
<tr>
<td>DEC</td>
<td>dst</td>
<td>Decrement</td>
</tr>
<tr>
<td>DECW</td>
<td>dst</td>
<td>Decrement Word</td>
</tr>
<tr>
<td>INC</td>
<td>dst</td>
<td>Increment</td>
</tr>
<tr>
<td>INCW</td>
<td>dst</td>
<td>Increment Word</td>
</tr>
<tr>
<td>SBC</td>
<td>dst, src</td>
<td>Subtract With Carry</td>
</tr>
<tr>
<td>SUB</td>
<td>dst, src</td>
<td>Subtract</td>
</tr>
</tbody>
</table>

### Logical Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand(s)</th>
<th>Name of Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>dst, src</td>
<td>Logical And</td>
</tr>
<tr>
<td>COM</td>
<td>dst</td>
<td>Complement</td>
</tr>
<tr>
<td>OR</td>
<td>dst, src</td>
<td>Logical Or</td>
</tr>
<tr>
<td>XOR</td>
<td>dst, src</td>
<td>Exclusive Or</td>
</tr>
</tbody>
</table>

### Program-Control Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand(s)</th>
<th>Name of Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>dst</td>
<td>Call</td>
</tr>
<tr>
<td>DJNZ</td>
<td>r, dst</td>
<td>Decrement and Jump If Nonzero</td>
</tr>
<tr>
<td>IRET</td>
<td></td>
<td>Interrupt Return</td>
</tr>
<tr>
<td>JP</td>
<td>cc, dst</td>
<td>Jump</td>
</tr>
<tr>
<td>JR</td>
<td>cc, dst</td>
<td>Jump Relative Return</td>
</tr>
<tr>
<td>RET</td>
<td></td>
<td>Return</td>
</tr>
</tbody>
</table>

APPENDIX A

Z8 Instruction Set: Functional Groups

Load Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand(s)</th>
<th>Name of Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR</td>
<td>dst</td>
<td>Clear</td>
</tr>
<tr>
<td>LD</td>
<td>dst, src</td>
<td>Load</td>
</tr>
<tr>
<td>LDC</td>
<td>dst, src</td>
<td>Load Constant Data</td>
</tr>
<tr>
<td>LDE</td>
<td>dst, src</td>
<td>Load External Data</td>
</tr>
<tr>
<td>POP</td>
<td>dst</td>
<td>Pop</td>
</tr>
<tr>
<td>PUSH</td>
<td>src</td>
<td>Push</td>
</tr>
</tbody>
</table>

Arithmetic Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand(s)</th>
<th>Name of Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>dst, src</td>
<td>Add With Carry</td>
</tr>
<tr>
<td>ADD</td>
<td>dst, src</td>
<td>Add</td>
</tr>
<tr>
<td>CP</td>
<td>dst, src</td>
<td>Compare</td>
</tr>
<tr>
<td>DA</td>
<td>dst</td>
<td>Decimal Adjust</td>
</tr>
<tr>
<td>DEC</td>
<td>dst</td>
<td>Decrement</td>
</tr>
<tr>
<td>DECW</td>
<td>dst</td>
<td>Decrement Word</td>
</tr>
<tr>
<td>INC</td>
<td>dst</td>
<td>Increment</td>
</tr>
<tr>
<td>INCW</td>
<td>dst</td>
<td>Increment Word</td>
</tr>
<tr>
<td>SBC</td>
<td>dst, src</td>
<td>Subtract With Carry</td>
</tr>
<tr>
<td>SUB</td>
<td>dst, src</td>
<td>Subtract</td>
</tr>
</tbody>
</table>

Logical Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand(s)</th>
<th>Name of Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>dst, src</td>
<td>Logical And</td>
</tr>
<tr>
<td>COM</td>
<td>dst</td>
<td>Complement</td>
</tr>
<tr>
<td>OR</td>
<td>dst, src</td>
<td>Logical Or</td>
</tr>
<tr>
<td>XOR</td>
<td>dst, src</td>
<td>Exclusive Or</td>
</tr>
</tbody>
</table>

Program-Control Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand(s)</th>
<th>Name of Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL</td>
<td>dst</td>
<td>Call</td>
</tr>
<tr>
<td>DJNZ</td>
<td>r, dst</td>
<td>Decrement and Jump If Nonzero</td>
</tr>
<tr>
<td>IRET</td>
<td></td>
<td>Interrupt Return</td>
</tr>
<tr>
<td>JP</td>
<td>cc, dst</td>
<td>Jump</td>
</tr>
<tr>
<td>JR</td>
<td>cc, dst</td>
<td>Jump Relative Return</td>
</tr>
<tr>
<td>RET</td>
<td></td>
<td>Return</td>
</tr>
</tbody>
</table>
### Bit-Manipulation Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Name of Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCM</td>
<td>dst, src</td>
<td>Test Complement Under Mask</td>
</tr>
<tr>
<td>TM</td>
<td>dst, src</td>
<td>Test Under Mask</td>
</tr>
<tr>
<td>AND</td>
<td>dst, src</td>
<td>Logical And</td>
</tr>
<tr>
<td>OR</td>
<td>dst, src</td>
<td>Logical Or</td>
</tr>
<tr>
<td>XOR</td>
<td>dst, src</td>
<td>Logical Exclusive Or</td>
</tr>
</tbody>
</table>

### Block-Transfer Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Name of Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDCI</td>
<td>dst, src</td>
<td>Load Constant Autoincrement</td>
</tr>
<tr>
<td>LDEI</td>
<td>dst, src</td>
<td>Load External Data Autoincrement</td>
</tr>
</tbody>
</table>

### Rotate and Shift Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand</th>
<th>Name of Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>RL</td>
<td>dst</td>
<td>Rotate Left</td>
</tr>
<tr>
<td>RLC</td>
<td>dst</td>
<td>Rotate Left Through Carry</td>
</tr>
<tr>
<td>RR</td>
<td>dst</td>
<td>Rotate Right</td>
</tr>
<tr>
<td>RRC</td>
<td>dst</td>
<td>Rotate Right Through Carry</td>
</tr>
<tr>
<td>SRA</td>
<td>dst</td>
<td>Shift Right Arithmetic</td>
</tr>
<tr>
<td>SWAP</td>
<td>dst</td>
<td>Swap Nibbles</td>
</tr>
</tbody>
</table>

### CPU Control Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operand</th>
<th>Name of Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCF</td>
<td></td>
<td>Complement Carry Flag</td>
</tr>
<tr>
<td>DI</td>
<td></td>
<td>Disable Interrupts</td>
</tr>
<tr>
<td>EI</td>
<td></td>
<td>Enable Interrupts</td>
</tr>
<tr>
<td>NOP</td>
<td></td>
<td>No Operation</td>
</tr>
<tr>
<td>RCF</td>
<td></td>
<td>Reset Carry Flag</td>
</tr>
<tr>
<td>SCF</td>
<td>src</td>
<td>Set Carry Flag</td>
</tr>
<tr>
<td>SRP</td>
<td>src</td>
<td>Set Register Pointer</td>
</tr>
</tbody>
</table>
A Comparison of Microcomputer Units

MAY 1981

INTRODUCTION

The microcomputer industry has recently developed single-chip microcomputers that incorporate on one chip functions previously performed by peripherals. These microcomputer units (MCUs) are aimed at markets requiring a dedicated computer. This report describes and compares the most powerful MCUs in today's market: the Zilog Z8611, the Intel 8051, and the Motorola MC6801. Table 1 lists facts that should be considered when comparing these MCUs.

Table 1. MCU Comparison

<table>
<thead>
<tr>
<th>FEATURES</th>
<th>Zilog Z8611</th>
<th>Intel 8051</th>
<th>Motorola MC6801</th>
</tr>
</thead>
<tbody>
<tr>
<td>On-Chip ROM</td>
<td>4Kx8</td>
<td>4Kx8</td>
<td>2Kx8</td>
</tr>
<tr>
<td>General-Purpose Registers</td>
<td>124</td>
<td>128</td>
<td>128</td>
</tr>
<tr>
<td>Special-Function Registers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Status/Control</td>
<td>16</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>I/O ports</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Parallel lines</td>
<td>32</td>
<td>32</td>
<td>29</td>
</tr>
<tr>
<td>Ports</td>
<td>Four 8-bit</td>
<td>Four 8-bit</td>
<td>Three 8-bit, one 5-bit</td>
</tr>
<tr>
<td>Handshake</td>
<td>Hardware on three ports</td>
<td>None</td>
<td>Hardware on one port</td>
</tr>
<tr>
<td>Interrupts</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Source</td>
<td>8</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>External source</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Vector</td>
<td>6</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>Priority</td>
<td>48 Programmable orders</td>
<td>2 Programmable orders</td>
<td>Nonprogrammable</td>
</tr>
<tr>
<td>Maskable</td>
<td>6</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>External Memory</td>
<td>120K bytes</td>
<td>124K bytes</td>
<td>64K bytes</td>
</tr>
<tr>
<td>Stack</td>
<td>16-Bit</td>
<td>8-Bit</td>
<td>16-Bit</td>
</tr>
<tr>
<td>Stack pointer</td>
<td>Yes, uses 8-bits</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Internal stack</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>External stack</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>FEATURES</td>
<td>Zilog Z8611</td>
<td>Intel 8051</td>
<td>Motorola MC6801</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------</td>
<td>------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Counter/</td>
<td>Two 8-bit</td>
<td>Two 16-bit</td>
<td>One 16-bit</td>
</tr>
<tr>
<td>Timers</td>
<td></td>
<td>or two 8-bit</td>
<td></td>
</tr>
<tr>
<td>Counters</td>
<td>Two 8-bit</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Prescalers</td>
<td>Two 6-bit</td>
<td>with 16-bits;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>with 8-bits</td>
<td></td>
</tr>
<tr>
<td>Addressing</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Modes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Register</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Indirect Register</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Indexed</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Direct</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Relative</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Immediate</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Implied</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Index Registers</td>
<td>124, Any</td>
<td>1, Uses the</td>
<td>1, Uses</td>
</tr>
<tr>
<td></td>
<td>general-</td>
<td>accumulator</td>
<td>16-bit index</td>
</tr>
<tr>
<td></td>
<td>purpose</td>
<td>register</td>
<td>register</td>
</tr>
<tr>
<td></td>
<td>register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Serial Communication</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Interface</td>
<td></td>
<td>One for each</td>
<td>One for both</td>
</tr>
<tr>
<td>Full duplex</td>
<td></td>
<td>Receiver</td>
<td>Transmitter/Receiver</td>
</tr>
<tr>
<td>UART</td>
<td></td>
<td>Receiver;</td>
<td></td>
</tr>
<tr>
<td>Interrupts</td>
<td></td>
<td>62.5K b/s</td>
<td>62.5K b/s</td>
</tr>
<tr>
<td>for transmit</td>
<td></td>
<td>187.5K b/s</td>
<td>12 MHz</td>
</tr>
<tr>
<td>and receive</td>
<td></td>
<td>@8 MHz</td>
<td>@4 MHz</td>
</tr>
<tr>
<td>Registers</td>
<td></td>
<td>93.5K b/s</td>
<td></td>
</tr>
<tr>
<td>Double buffer</td>
<td></td>
<td>@12 MHz</td>
<td></td>
</tr>
<tr>
<td>Serial Data Rate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Speed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instruction</td>
<td>2.2 Usec</td>
<td>1.5 Usec</td>
<td>3.9 Usec</td>
</tr>
<tr>
<td>execution average</td>
<td>1.5 Usec @12 MHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Longest instruction</td>
<td>4.25 Usec</td>
<td>4 Usec</td>
<td>10 Usec</td>
</tr>
<tr>
<td></td>
<td>2.8 Usec @12 MHz</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clock Frequency</td>
<td>8 and 12 MHz</td>
<td>12 MHz</td>
<td>4 MHz</td>
</tr>
<tr>
<td>Power Down Mode</td>
<td>Saves first</td>
<td>Saves first</td>
<td>Saves first</td>
</tr>
<tr>
<td></td>
<td>128 registers</td>
<td>128 registers</td>
<td>64 registers</td>
</tr>
<tr>
<td>Context Switching</td>
<td>Saves PC</td>
<td>Saves PC;</td>
<td>Saves PC, PSW,</td>
</tr>
<tr>
<td></td>
<td>and flags</td>
<td>programmer</td>
<td>accumulators,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>must save all</td>
<td>and Index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>registers</td>
<td>register</td>
</tr>
</tbody>
</table>
ARCHITECTURAL OVERVIEW

This section examines three chips: the on-chip functions and data areas manipulated by the Zilog, Intel and Motorola MCUs. The three chips have somewhat similar architectures. There are, however, fundamental differences in design criteria. The 8051 and the MC6801 were designed to maintain compatibility with older products, whereas the Z8611 design was free from such restrictions and could experiment with new ideas. Because of this, the accumulator architectures of the MC6801 and the 8051 are not as flexible as that of the Z8611, which allows any register to be used as an accumulator.

Memory Spaces

The Z8611 CPU manipulates data in four memory spaces:

- 60K bytes of external data memory
- 60K bytes of external program memory
- 4K bytes of internal program memory (ROM)
- 144-byte register file

The 8051 CPU manipulates data in four memory spaces:

- 64K bytes of external data memory
- 60K bytes of external program memory
- 4K bytes of internal program memory
- 148-byte register file

The MC6801 manipulates data in three memory spaces:

- 62K bytes of external memory
- 2K bytes of internal program memory
- 149-byte register file

On-Chip ROM. All three chips have internal ROM for program memory. The Z8611 and the 8051 have 4K bytes of internal ROM, and the MC6801 has 2K bytes. In some cases, external memory may be required with the MC6801 that is not necessary with the Z8611 or the 8051.

On Chip RAM. All three chips use internal RAM as registers. These registers are divided into two categories: general-purpose registers and special function registers (SFRs).

The 124 general-purpose registers in the Z8611 are divided into eight groups of 16 registers each. In the first group, the lowest four registers are the I/O port registers. The other registers are general purpose and can be accessed with an 8-bit address or a short 4-bit address. Using the 4-bit address saves bytes and execution time. Four-bit short addresses are discussed later. The general-purpose registers can be used as accumulators, address pointers, or Index registers.

The 128 general-purpose registers in the 8051 are grouped into two sets. The lower 32 bytes are allocated as four 8-register banks, and the upper registers are used for the stack or for general purpose. The registers cannot be used for indexing or as address pointers.

The MC6801 also has a 128-byte, general-purpose register bank, which can be used as a stack or as address pointers, but not as Index registers.

As pointed out in Table 1, any of the Z8611 general-purpose registers can be used for indexing; the MC6801 and the 8051 cannot use registers this way. The Z8611 can use any register as an accumulator; the MC6801 and the 8051 have fixed accumulators. The use of registers as memory pointers is very valuable, and only the Z8611 can use its registers in this way.

The number of general-purpose registers on each chip is comparable. However, because of its flexible design, the Z8611 clearly has a more powerful register architecture.
The Z8611 has 20 special function registers used for status, control, and I/O. These registers include:

- Two registers for a 16-bit Stack Pointer (SPH, SPL)
- One register used as Register Pointer for working registers (RP)
- One register for the status flags (FLAGS)
- One register for interrupt priority (IPR)
- One register for interrupt mask (IMR)
- One register for interrupt request (IRQ)
- Three mode registers for the four ports (P01M, P2M, P3M)
- Serial communications port used like a register (SIO)
- Two counter/timer registers (TO, T1)
- One Timer Mode Register (TMR)
- Two prescaler registers (PRE0, PRE1)
- Four I/O ports accessed as registers (PORT0, PORT1, PORT2, PORT3)

The 8051 also has 20 special function registers used for status, control, and I/O. They include:

- One register for the Stack Pointer (SP)
- Two accumulators (A, B)
- One register for the Program Status Word (PSW)
- Two registers for pointing to data memory (DPH, DPL)
- Four registers that serve as two 16-bit counter/timers (TH0, TH1, TL0, TL1)
- One mode register for the counter/timers (TMOD)
- One control register for the counter/timers (TCON)
- One register for interrupt enable (IEC)
- One register for interrupt priority (IPC)
- One register for serial communications buffer (SBUF)
- One register for serial communications control (SCON)
- Four registers used as the four I/O ports (PO, P1, P2, P3)

The MC6801 has 21 special function registers used for status, control, and I/O. These include:

- One register for RAM/EROM control
- One serial receive register
- One serial transmit register
- One register for serial control and status
- One serial rate and mode register
- One register for status and control of port 3
- One register for status and control of the timer
- Two registers for the 16-bit timer
- Two registers for 16-bit input capture used with timer
- Two registers for 16-bit output compare used with timer
- Four data direction registers associated with the four I/O ports
- Four I/O ports

The special function registers in the three chips seem comparable in number and function. However, upon closer examination, the SFRs of the MC6801 prove less efficient than those of the Z8611. The MC6801 has five registers associated with the I/O ports, whereas the Z8611 uses only three registers for the same functions. The MC6801 uses four registers to perform the serial communication function, whereas the Z8611 uses only one register and part of another.

The 8051 uses two registers for the accumulators; the Z8611 is not limited by this restriction. The 8051 also uses two registers for the serial communication interface, whereas the Z8611 accomplishes the same job with one register. Another two registers in the 8051 are used for data pointers; these are not necessary in the Z8611 since any register can be used as an address pointer.

The Z8611 uses registers more efficiently than either the MC6801 or the 8051. The registers saved by this optimal design are used to perform the functions needed for enhanced interrupt handling and for register pointing with short addresses. The Z8611 also supplies the extra register required for the external stack. These features are not available on the 8051 or the MC6801.

External Memory. All three chips can access external memory. The Z8611 and the 8051 can generate signals used for selecting either program or data memory. The Data Memory strobe (the signal used for selecting data or program memory) gives the Z8611 access to 120K bytes of external memory (60K bytes in both program and data memory). The 8051 can use 124K bytes of external memory (64K bytes of external data memory and 60K bytes of external program memory). The MC6801 can access only 62K bytes of external memory and does not distinguish between program and data memory. Thus, the Z8611 and the 8051 are clearly able to access more external memory than the MC6801.

On-Chip Peripheral Functions

In addition to the CPU and memory spaces, all chips provide an interrupt system and extensive I/O facilities including I/O pins, parallel I/O ports, a bidirectional address/data bus, and a serial port for I/O expansion.

Interrupts. The Z8611 acknowledges interrupts from eight sources, four are external from pinsIRQ0-IRQ3, and four are internal from serial-in, serial-out, and the two counter/timers. All interrupts are maskable, and a wide variety of priorities are realized with the Interrupt Mask Register and the Interrupt Priority Registers (see Table 1). All Z8611 interrupts are vectored, with six vectors located in the on-chip ROM. The vectors are fixed locations, two bytes long, that contain the memory address of the service routine.
The 8051 acknowledges interrupts from five sources: two external sources (from INTO and INT1) and three internal sources (one from each of the internal counters and one from the serial I/O port). All interrupts can be disabled individually or globally. Each of the five sources can be assigned one of two priorities: high or low. All 8051 interrupts are vectored. There are five fixed locations in memory, each eight bytes long, allocated to servicing the interrupt.

The MC6801 has one external interrupt, one non-maskable interrupt, an internal interrupt request, and a software interrupt. The internal interrupts are caused by the serial I/O port, timer overflow, timer output compare, and timer input capture. The priority of each interrupt is preset and cannot be changed. The external interrupt can be masked in the Condition Code register. The MC6801 vectors the interrupts to seven fixed addresses in ROM where the 16-bit address of the service routine is located.

When an interrupt occurs in the 8051, only the Program Counter is saved; the user must save the flags, accumulator, and any registers that the interrupt service routine might affect. The MC6801 saves the Program Counter, accumulators, Index register, and the PSW; the user must save all registers that the interrupt service routine might affect. The Z8611 saves the Program Counter and the Flags register. To save the 16 working registers, only the Register Pointer register need be pushed onto the stack and another set of working registers is used for the service routine. For more detail on working registers and interrupt context switching, see the Z8 Technical Manual (03-3047-02).

With regard to interrupts, the Z8611 is clearly superior. The Z8611 requires only one command to save all the working registers, which greatly increases the efficiency of context switching.

I/O Facilities. The Z8611 has 32 lines dedicated to I/O functions. These lines are grouped into four ports with eight lines per port. The ports can be configured individually under software control to provide input, output, multiplexed address/data lines, timing, and status. Input and output can be serial or parallel, with or without handshake. One port can be configured for serial transmission and four ports can be configured for parallel transmission. With parallel transmission, ports 0, 1, and 2 can transmit data with the handshake provided by port 3.

The 8051 also has 32 I/O lines grouped together into four ports of eight lines each. The ports can be configured under program control for parallel or serial I/O. The ports can also be configured for multiplexed address/data lines, timing, and status. Handshake is provided by user software.

The MC6801 has 29 lines for I/O (three 8-bit ports and one 5-bit port). One port has two lines for handshake. The ports provide all the signals needed to control input and output either serially or in parallel, with or without multiplexed address/data lines. They can be used to interface with external memory.

The main differences in I/O facilities are the number of 8-bit ports and the hardware handshake. The Z8611 and the 8051 have four 8-bit ports, whereas the MC6801 has three 8-bit ports and an additional 5-bit port. The Z8611 has hardware handshake on three ports, the MC6801 has hardware handshake on only one port, and the 8051 has no hardware handshake.

Counter/timers. The Z8611 has two 8-bit counters and two 6-bit programmable prescalers. One prescaler can be driven internally or externally; the other prescaler is driven internally only. Both timers can interrupt the CPU when counting is completed. The counters can operate in one of two modes: they can count down until interrupted or they can count down, reload the initial value, and start counting down again (continuously). The counters for the Z8611 can be used for measuring time intervals and pulse widths, generating variable pulse widths, counting events, or generating periodic interrupts.

The 8051 has two 16-bit counter/timers for measuring time intervals and pulse widths, generating pulse widths, counting events, and generating periodic interrupts. The counter/timers have several modes of operation. They can be used as 8-bit counters or timers with two 5-bit programmable prescalers. They can also be used as 16-bit counter/timers. Finally, they can be set as 8-bit modulo-n counters with the reload value held in the high byte of the 16-bit register. An interrupt is generated when the counter/timer has completed counting.

The MC6801 has one 16-bit counter which can be used for pulse-width measurement and generation. The counter/timer actually consists of three 16-bit registers and an 8-bit control/status register. The timer has an input capture register, an output compare register, and a free-running counter. All three 16-bit registers can generate interrupts.

Serial Communications Interface. The Z8611 has a programmable serial communication interface. The chip contains a UART for full-duplex, asynchronous, serial receiver/transmitter operation. The bit rate is controlled by counter/timer 0 and has a maximum bit rate of 93,500 b/s. An interrupt is generated when an assembled character is transferred to the receive buffer. The transmitted character generates a separate interrupt. The receive register is double-buffered. A hardware parity generator and detector are optional.

The 8051 handles serial I/O using one of its parallel ports. The 8051 bit rate is controlled
by counter/timer 1 and has a maximum bit rate of 187,500 b/s. The 8051 generates one interrupt for both transmission and receipt. The receive register is double-buffered.

The MC6801 contains a full-duplex, asynchronous, serial communication interface. The bit rate is controlled by a rate register and by the MCU's clock or an external clock. The maximum bit rate is 62,500 b/s. Both the transmit and the receive registers are double-buffered. The MC6801 generates only one interrupt for both transmit and receive operations. No hardware parity generation or detection is available, although it does have automatic detection of framing errors and overrun conditions.

The 8051 and the MC6801 generate only one interrupt for both transmit and receive, whereas the Z8611 has a separate interrupt for each. The ability to generate separate interrupts greatly enhances the use of serial communications, since separate service routines are often required for transmitting and receiving.

Other differences between the Z8611, MC6801, and the 8051 occur in the hardware parity detector, the double-buffering of registers, framing error detectors and overrun conditions. The 8051 has a faster data rate than either the Z8611 or the MC6801. The MC6801 has the advantage of a hardware framing error detector and automatic detection of overrun conditions. The MC6801 also has both its transmit and receive registers double-buffered. The Z8611 has a hardware parity detector. For detection of framing errors and overrun conditions, a simple, low-overhead software check is available that uses only two instructions. See Z8600 Software Framing Error Detection Application Brief (document #617-1881-0004).

**INSTRUCTION ARCHITECTURE**

The architecture of the Z8611 is designed specifically for microcomputer applications. This fact is manifest in the instruction composition. The arduous task of programming the MC6801 and the 8051 starkly contrasts that of programming the Z8611.

**Addressing Modes**

The Z8611 and the 8051 both have six addressing modes: Register, Indirect Register, Indexed, Direct, Relative, and Immediate. The MC6801 has five addressing modes: Accumulator, Indexed, Direct, Relative, and Immediate. A quick comparison of these addressing modes reveals the versatility of the Z8611 and the 8051. The addressing modes of the MC6801 have several restrictions, as shown in Table 1. While the 8051 has all the addressing modes of the Z8611, its use of them is restricted. The Z8611 allows many more combinations of addressing modes per instruction, because any of its registers can be used as an accumulator. For example, the instructions to clear, complement, rotate, and swap nibbles are all accumulator oriented in the 8051 and operate on the accumulator only. These same commands in the Z8611 can use any register and access it either directly, with register addressing, or with indirect register addressing.

**Indexed Addressing.** All three chips differ in their handling of indexing. The Z8611 can use any register for indexing. The 8051 can use only the accumulator as an Index register in conjunction with the data pointer or the Program Counter. The MC6801 has one 16-bit Index register. The address located in the second byte of an instruction is added to the lower byte of the Index register. The carry is added to the upper byte for the complete address. The MC6801 requires the index value to be an immediate value.

The MC6801 has only one 16-bit Index register and an immediate 8-bit value from the second byte of the instruction. Hence, the Indexed mode of the MC6801 is much more restrictive than that of the Z8611. The 8051 must use the accumulator as its only Index register, loading the accumulator with the register address each time a reference is made. Then, using indexing, the data is moved into the accumulator, eradicating the previous index. This forces a stream of data through the accumulator and requires a reload of the index before access can be made again. The Z8611 is clearly superior to both the MC6801 and the 8051 in the flexibility of its indexed addressing mode.

**Short and Long Addressing.** Short addressing helps to optimize memory space and execution speed. In sample applications of short register addressing, an eight percent decrease in the number of bytes used was recorded.

All three chips have short addressing modes, but the Z8611 has short addressing for both external memory and register memory. The 8051 has short addressing for the lowest 32 registers only.

The Z8611 has two different modes for register addressing. The full-byte address can be used to provide the address, or a 4-bit address can be used with the Register Pointer. To use the working registers, the Register Pointer is set for a particular bank of 16 registers, and then one of the 16 registers is addressed with four bits. Another feature for addressing external memory is the use of a 12-bit address in place of a full 16-bit address. To use the 12-bit address, one port supplies the eight multiplexed address/data lines and another port supplies four bits for the address. The remaining four bits of the second port can be used for 1/0. This feature allows access to a maximum of 10K bytes of memory.
The 8051 uses short addresses by organizing its lowest 32 registers into four banks. The bank select is located in a 2-bit field in the PSW, with three bits addressing the register in the bank.

The MC6801 uses extended addressing for addressing external memory. With a special, nonmultiplexed expansion mode, 256 bytes of external memory can be accessed without the need for an external address latch. The MC6801 uses one 8-bit port for the address and another port for the data.

**Stacks**

The Z8611 and the MC6801 provide for external stacks, which require a 16-bit Stack Pointer. Internal stacks use only an 8-bit Stack Pointer. The 8051 uses only a limited internal stack requiring an 8-bit Stack Pointer. Using an external stack saves the internal RAM registers for general-purpose use.

**Summary**

The stack structure of the Z8611 and the MC6801 is better than that of the 8051. In most applications, the 8051 is more flexible and easier to program than the MC6801. The Z8611 is easier to use than either the 8051 or the MC6801 because of its register flexibility and its numerous combinations of addressing modes. The 8051 features a unique 4xn multiply and divide command. The MC6801 has a multiply, but it takes 10-4s to perform it.

In summary, the Z8611 has the most flexible addressing modes, the most advanced indexing capabilities, and superior space- and time-saving abilities with respect to short addressing.

**DEVELOPMENT SUPPORT**

All three vendors provide development support for their products. This section discusses the different support features, including development chips, software, and modules.

**Chips**

Zilog offers an entire family of microcomputer chips for product development and final product. The Z8611 is a single-chip microcomputer with 4K bytes of mask-programmed ROM. For development, two other chips are offered. The Z8612 is a 64-pin, development version with full interface to external memory. The Z8613 is a prototype version that uses a functional, piggy-back, EPROM protopak. The Z8613 can use either a 4K EPROM (2732) or a 2K EPROM (2716). Zilog also offers a ROMless version in a 40-pin package that has all the features of the Z8611 except on-board ROM (Z8681).

Intel offers a similar line of development chips with its 8051 family. The 8031 has no internal ROM and the 8751 has 4K of internal EPROM.

Motorola offers the MC6801, MC6803, MC6803NR, and MC68701. These are all similar except the MC68701 has 2K bytes of EPROM and the MC6801 has 2K bytes of ROM. The MC6803 has no internal ROM and the MC6803NR has neither ROM nor RAM on board.

The Z8613 and the MC68701 are both available now, but the 8751 is still unavailable (as of April 1981).

**Software**

Development software includes assemblers, and conversion programs. All manufacturers offer some or all of these features.

Since the MC6801 is compatible with the 8080, there is no need for a new assembler. The Z8611 and the 8051 both offer assemblers for their products. The Zilog PLZ/ASM assembler generates relocatable and absolute object code. PLZ/ASM also supports high-level control and data statements, such as IF... THEN... ELSE. Intel offers an absolute macroassembler, ASM51, with their product. They also offer a program for converting 8048 code to 8051 code.

**Modules**

The Z8611 development module has two 64-pin development versions of the 40-pin, ROM-masked Z8611. Intel offers the EM-51 emulation board, which contains a modified 8051 and PROM or EPROM in place of memory. Motorola has the MEX6801EVM evaluation board for program development. All three development boards are available now.

**ADDITIONAL FEATURES**

Additional features include Power Down mode, self-testing, and family-compatibility.

**Power Down Mode**

All three microcomputers offer a Power Down mode. The Z8611 and the 8051 save all of their registers with an auxiliary power supply. The MC6801 uses an auxiliary power supply to save only the first 64 bytes of its register file.

The Z8611 uses one of the crystal input pins for the external power supply to power the registers in Power Down mode. Since the XTAL2 input must be used, an external clock generator is necessary and is input via XTAL1. The 8051 and the MC6801 both have an input reserved for this function. The MC6801 uses the Vcc standby pin, and the 8051 uses the Vpd pin.
Family Compatibility

Another strength of the Z8611 is its expansion bus, which is completely compatible with the Zilog Z-BUS™. This means that all Z-BUS peripherals can be used directly with the Z8611.

The MC6801 is fully compatible with all MC6800 family products. The 8051 is software compatible with the older 8048 series and all others in that family.

BENCHMARKS

The following benchmark tests were used in this report to compare the Z8611, 8051, and MC6801:

- Generate CRC check for 16-bit word.
- Search for a character in a block of memory.
- Execute a computed GOTO - jump to one of eight locations depending on which of the eight bits is set.
- Shift a 16-word five places to the right.
- Move a 64-byte block of data from external memory to the register file.
- Toggle a single bit on a port.
- Measure the subroutine overhead time.

These programs were selected because of their importance in microcomputer applications. Algorithms that reflect a unique function or feature were excluded for the sake of comparison. Although programs can be optimized for a particular chip and for a particular attribute (code density or speed) these programs were not.

The figures cited in this text are taken directly from the vendor's documentation. Therefore, the cycles given below for the MC6801 and the 8051 are in machine cycles and the Z8611 figures are given in clock cycles. The Z8611 clock cycles should be divided by six to give the instruction time in microseconds. The 8051 and MC6801 machine cycle is 1.3 µs, and the Z8611 clock cycle is 166.67 µs at 12 MHz.

Because of the lack of availability of the MC6801 and the 8051, the benchmark programs listed here have not yet been run. When these products are readily available, the programs will be run and later editions of this document will reflect any changes in the findings.

Program Listings

**Z8051**

<table>
<thead>
<tr>
<th>Machine Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV INDEX, #8</td>
<td>1</td>
</tr>
<tr>
<td>LOOP: MOV A, DATA</td>
<td>1</td>
</tr>
<tr>
<td>XRL A, HCHECK</td>
<td>1</td>
</tr>
<tr>
<td>RLC A</td>
<td>1</td>
</tr>
<tr>
<td>MOV A, LCHECK</td>
<td>1</td>
</tr>
<tr>
<td>XRL A, LPOLY</td>
<td>1</td>
</tr>
<tr>
<td>RLC A</td>
<td>1</td>
</tr>
<tr>
<td>MOV LCHECK, A</td>
<td>1</td>
</tr>
<tr>
<td>MOV A, HCHECK</td>
<td>1</td>
</tr>
<tr>
<td>XRL A, HPOLY</td>
<td>1</td>
</tr>
<tr>
<td>RLC A</td>
<td>1</td>
</tr>
<tr>
<td>MOV HCHECK, A</td>
<td>1</td>
</tr>
<tr>
<td>CLR C</td>
<td>1</td>
</tr>
<tr>
<td>MOV A, DATA</td>
<td>1</td>
</tr>
<tr>
<td>RLC A</td>
<td>1</td>
</tr>
<tr>
<td>MOV DATA, A</td>
<td>1</td>
</tr>
<tr>
<td>DJNZ INDEX, LOOP</td>
<td>2</td>
</tr>
<tr>
<td>RET</td>
<td>2</td>
</tr>
</tbody>
</table>

N = $3 \times 17 \times 8 = 139$ cycles
©12 MHz = 139 µs
Instructions = 18
Bytes = 31

**MC6801**

<table>
<thead>
<tr>
<th>Machine Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAA #$0B</td>
<td>2</td>
</tr>
<tr>
<td>LOOP: STAA COUNT</td>
<td>3</td>
</tr>
<tr>
<td>LDAA HCHECK</td>
<td>3</td>
</tr>
<tr>
<td>EORA DATA</td>
<td>3</td>
</tr>
<tr>
<td>ROLA</td>
<td>2</td>
</tr>
<tr>
<td>LDAD POLY</td>
<td>4</td>
</tr>
<tr>
<td>EORA HCHECK</td>
<td>3</td>
</tr>
<tr>
<td>EORB LCHECK</td>
<td>3</td>
</tr>
<tr>
<td>ROLB</td>
<td>2</td>
</tr>
<tr>
<td>ROLA</td>
<td>2</td>
</tr>
<tr>
<td>STAD LCHECK</td>
<td>4</td>
</tr>
<tr>
<td>ASL DATA</td>
<td>6</td>
</tr>
<tr>
<td>DEC COUNT</td>
<td>6</td>
</tr>
<tr>
<td>BNE LOOP</td>
<td>4</td>
</tr>
<tr>
<td>RTS</td>
<td>5</td>
</tr>
</tbody>
</table>

N = $45 \times 8 + 7 = 367$ cycles
©4 MHz = 367 µs
Instructions = 15
Bytes = 28

**Z8611**

<table>
<thead>
<tr>
<th>Clock Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD INDEX, #8</td>
<td>6</td>
</tr>
<tr>
<td>LOOP: LD R6, DATA</td>
<td>6</td>
</tr>
<tr>
<td>XOR R6, HCHECK</td>
<td>6</td>
</tr>
<tr>
<td>RLC R6</td>
<td>6</td>
</tr>
<tr>
<td>XOR LCHECK, LPOLY</td>
<td>6</td>
</tr>
<tr>
<td>RLC LCHECK</td>
<td>6</td>
</tr>
<tr>
<td>XOR HCHECK, HPOLY</td>
<td>6</td>
</tr>
<tr>
<td>RLC HCHECK</td>
<td>6</td>
</tr>
<tr>
<td>RCF</td>
<td>6</td>
</tr>
<tr>
<td>RLC DATA</td>
<td>6</td>
</tr>
<tr>
<td>DJNZ INDEX, LOOP</td>
<td>12</td>
</tr>
<tr>
<td>RET</td>
<td>14</td>
</tr>
</tbody>
</table>

N = $20 + 66 \times 7 + 64 = 546$ cycles
©12 MHz = 91 µs
Instructions = 12
Bytes = 22
Character Search Through Block of 40 Bytes

**8051**

<table>
<thead>
<tr>
<th>Machine</th>
<th>Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV INDEX, #41</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>MOV DPTR, #TABLE</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>LOOP1: DJNZ INDEX, LOOP 2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>SJMP OUT</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>LOOP2: MOV A, INDEX</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>MOV A, @A+DPTR</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>CJNE A, CHARAC, LOOP1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>OUT:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ N = 3 + 39 \times 7 + 4 = 280 \text{ cycles} \]

@12 MHz = 280 µs

Instructions = 7

Bytes = 15

**MC6801**

<table>
<thead>
<tr>
<th>Machine</th>
<th>Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAB #$40</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>LDAA #CHARAC</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>LDX #TABLE</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>LOOP: CMPA $0, X</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>BEQ OUT</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>INX</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>DECB</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>BNE LOOP</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>OUT:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ N = 7 + 40 \times 17 = 687 \text{ cycles} \]

@4 MHz = 687 µs

Instructions = 8

Bytes = 15

**Z8611**

<table>
<thead>
<tr>
<th>Machine</th>
<th>Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD INDEX, #40</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>LOOP: LD DATA, TABLE (INDEX)</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>CP DATA, CHARAC</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>JR Z, OUT</td>
<td>12 or 10</td>
<td>2</td>
</tr>
<tr>
<td>DJNZ INDEX, LOOP</td>
<td>12 or 10</td>
<td>2</td>
</tr>
<tr>
<td>OUT:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ N = 6 + 38 \times 40 = 1524 \text{ cycles} \]

@12 MHz = 254 µs

Instructions = 5

Bytes = 11

Shift 16-Bit Word to Right 5-Bits

**8051**

<table>
<thead>
<tr>
<th>Machine</th>
<th>Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV INDEX, #5</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>LOOP: CLR C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOV A, WORD + 1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>RRC A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MOV WORD + 1, A</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>MOV A, WORD</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>RRC A</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MOV WORD, A</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>DJNZ INDEX, LOOP</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

\[ N = 1 + 9 \times 5 = 46 \text{ Cycles} \]

@12 MHz = 46 µs

Instructions = 9

Bytes = 15

**MC6801**

<table>
<thead>
<tr>
<th>Machine</th>
<th>Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDX #5</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>LDAD WORK</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>LOOP: LSRD</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>DEX</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>BNE LOOP</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>STAD WORD</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

\[ N = 10 \times 5 + 11 = 61 \text{ Cycles} \]

@4 MHz = 61 µs

Instructions = 6

Bytes = 11

**Z8611**

<table>
<thead>
<tr>
<th>Machine</th>
<th>Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD INDEX, #5</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>LOOP: CCF</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>RRC WORD + 1</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>RRC WORD</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>DJNZ INDEX, LOOP</td>
<td>12 or 10</td>
<td>2</td>
</tr>
</tbody>
</table>

\[ N = 6 + 4 \times 30 + 28 = 154 \text{ Cycles} \]

@12 MHz = 26 µs

Instructions = 5

Bytes = 9

751-1534-0002 1-23 4-23-81
### Computed GOTO

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Machine</th>
<th>Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV INDEX, #40</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>LOOP: MOV A, DATA</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>RLC A</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>JC OUT</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>MOV A, INDEX</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ADD A, #3</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>MOV INDEX, A</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SJMP LOOP</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>OUT: MOV DPTR, #TABLE</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>MOV A, INDEX</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>JMP @A+DPTR</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**Table: LCALL ADDR1**

- LCALL ADDR2
  - N = 1+9x7+11 = 75 Cycles
  - @12 MHz = 75 μs
  - Instructions = 12
  - Bytes = 21

### Move 64-Byte Block

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Machine</th>
<th>Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV INDEX, #COUNT</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>LOOP: MOV DPTR, #ADDR1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>MOV A, @DPTR</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>INC #ADDR1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>MOV @ADDR2,A</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>INC ADDR2</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>DJNZ INDEX, LOOP</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>N = 1+9x64 = 577 Cycles</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**MC6801**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Machine</th>
<th>Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAB #COUNT</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>LOOP: LDX ADDR1</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>LDAA 0, X</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>INX</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>STAA ADDR1</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>LDX ADDR2</td>
<td>4</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>STAA 0, X</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>INX</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>STX ADDR2</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>DECX</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>N = 64x36+2 = 2306 Cycles</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>@4 MHz = 2306 μs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instructions = 11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bytes = 21</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### ZB611

**Clock**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Machine</th>
<th>Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR INDEX</td>
<td>6</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>LOOP: INC INDEX</td>
<td>6</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>RLC DATA</td>
<td>6</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>JR NC, LOOP</td>
<td>12 or 10</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>LD ADDR, TABLE 1, (INDEX)</td>
<td>10</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>LD ADDR+1, TABLE 2, (INDEX)</td>
<td>10</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>JP @ADDR</td>
<td>12</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>N = 6+24x7+54 = 228 Cycles</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>@12 MHz = 38 μs</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instructions = 7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bytes = 15</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**Notes:**

- 751-1534-0002
- 1-24
- 4-23-81
Toggle a Port Bit

Subroutine Call/Return Overhead

<table>
<thead>
<tr>
<th>8051</th>
<th>Machine Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>XRL</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>PO, #YY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N = 2 Cycles</td>
<td></td>
<td></td>
</tr>
<tr>
<td>@12 MHz = 2 μs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instructions = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bytes = 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8051</th>
<th>Machine Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCALL</td>
<td>SUBR</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MC6801</th>
<th>Machine Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAA PORTO</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>EDRA #YY</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>STAA PORTO</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>N = 8 Cycles</td>
<td></td>
<td></td>
</tr>
<tr>
<td>@4 MHz = 8 μs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instructions = 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bytes = 6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MC6801</th>
<th>Machine Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSR SUBR</td>
<td>9</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Z8611</th>
<th>Clock Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>XOR PORTO, #YY</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>N = 10 Cycles</td>
<td></td>
<td></td>
</tr>
<tr>
<td>@12 MHz = 1.7 μs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instructions = 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Byte = 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Z8611</th>
<th>Clock Cycles</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CALL @SUBR</td>
<td>20</td>
<td>2</td>
</tr>
</tbody>
</table>

Results

Table 2 summarizes the results of this comparison. The relative performance column lists the speeds of the MC6801 and 8051 divided by the Z8611 speeds (12 MHz). The overall performance averages the separate relative performances. The higher the number, the faster the Z8611 as compared to the MC6801 and the 8051.

The relative performance figures show that the Z8611 runs 50 percent faster than the 8051 and 250 percent faster than the MC6801. Although speed is not necessarily the most important criterion for selecting a particular product, the Z8611 proves to be an undeniably superior product when speed is added to the advantages of programming ease, code density, and flexibility.
Table 2. Benchmark Program Results

<table>
<thead>
<tr>
<th>Benchmark Test</th>
<th>MC6801 (4 MHz) cycles time</th>
<th>8051 (12 MHz) cycles time</th>
<th>Z8 (8 MHz) cycles time</th>
<th>Z8 (12 MHz) cycles time</th>
<th>Relative Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC Generation</td>
<td>367</td>
<td>139</td>
<td>546</td>
<td>546</td>
<td>4.03 1.53</td>
</tr>
<tr>
<td>Character Search</td>
<td>687</td>
<td>280</td>
<td>1524</td>
<td>1524</td>
<td>2.70 1.10</td>
</tr>
<tr>
<td>Computed GOTO</td>
<td>110</td>
<td>75</td>
<td>228</td>
<td>228</td>
<td>2.89 1.97</td>
</tr>
<tr>
<td>Shift Right 5 Bits</td>
<td>61</td>
<td>46</td>
<td>154</td>
<td>154</td>
<td>2.35 1.78</td>
</tr>
<tr>
<td>Move 64-byte block</td>
<td>2306</td>
<td>577</td>
<td>1924</td>
<td>1924</td>
<td>7.18 1.80</td>
</tr>
<tr>
<td>Subroutine Overhead</td>
<td>14</td>
<td>4</td>
<td>34</td>
<td>34</td>
<td>2.46 0.70</td>
</tr>
<tr>
<td>Toggle a Port Bit</td>
<td>8</td>
<td>2</td>
<td>10</td>
<td>10</td>
<td>4.71 1.18</td>
</tr>
</tbody>
</table>

Note: All times are given in microseconds.

Table 3. Byte/Instruction/Time Comparison

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Instructions</th>
<th>Time (microseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC6801</td>
<td>8051</td>
<td>Z8611</td>
</tr>
<tr>
<td>-------</td>
<td>--------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>CRC Generation</td>
<td>28</td>
<td>31</td>
</tr>
<tr>
<td>Character Search</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>Shift Right 5 Bits</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>Computed GOTO</td>
<td>17</td>
<td>21</td>
</tr>
<tr>
<td>Move Block</td>
<td>21</td>
<td>10</td>
</tr>
<tr>
<td>Toggle Port Bit</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>Subroutine Call</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
SUMMARY

The hardware of the three chips compared is very similar. The Z8611, however, has several advantages, the most important of which is its interrupt structure. It is more advanced than the interrupt structures of both the 8051 and the MC6801. Other advantages of the Z8611 over either the MC6801 or the 8051 include I/O facilities with parity detection and hardware handshake and a larger amount of internal ROM (the MC6801 has only 2K bytes).

Substantial differences are apparent with regard to software architecture. The addressing modes of the Z8611 are more flexible than those of either the MC6801 or the 8051. The Z8611 can use byte-saving addressing with working registers, and it has short external addresses for saving I/O lines. It can also provide for an external stack. The register architecture (as opposed to the accumulator architecture) of the Z8611 saves execution time and enhances programming speed by reducing the byte count.

The Z8611 microcomputer stands out as the most powerful chip of the three, and concurrently, it is the easiest to program and configure.
The Interrupt Request Register (IRQ, R250) stores requests from the six possible interrupt sources (IRQ0-IRQ5) in the Z8600 series microcomputer. In addition to other functions, a hardware reset to the Z8600 disables the IRQ register and resets its request bits. Before the IRQ will register requests, it must first be enabled by executing an Enable Interrupts (EI) instruction. Setting the Enable Interrupt bit in the Interrupt Mask Register (IMR, R251) is not an equivalent operation for this purpose; to enable the IRQ, an EI instruction is required. The function of this EI instruction is distinct from its task of globally enabling the interrupt system. Even in a polled system where IRQ bits are tested in software, it is necessary to execute the EI.

The designer must ensure that unexpected and undesirable interrupt requests will not occur after the EI is executed. One method of doing this is to reset all interrupt enable bits in the IMR for levels that are possible interrupt sources; the EI instruction may then be safely executed. Once EI is executed, the program may immediately execute a Disable Interrupts (DI) instruction. The code necessary to perform these operations is as follows:

```
RESET: LD IMR, #%XX ;SET INTERRUPT MASK!
        EI                ;ENABLE GLOBAL INTERRUPT, ENABLE IRQ!
```

where XX has a Ø in each bit position corresponding to the interrupt level to be disabled. If all IMR bits are to be reset, a CLR IMR instruction may be used.

![Figure 1 - IRQ Reset Functional Logic Diagram](image-url)
12-Bit Addressing with the Z8 Family

Application Brief

January 1981

12-BIT ADDRESSING WITH THE Z8600 SERIES FAMILY

The Z8601 can manipulate data in four memory spaces: internal program memory, internal register file, external program memory, and external data memory. The internal register file is not discussed in this paper. Port 3 may be configured optionally to provide a Data Memory (DM) strobe that is used to select program and data memory. The Z8601 generates another signal, Data Strobe (DS), that signals an external memory operation. DS is generated each time an address greater than 2047 is used.

The Z8601 has 2K bytes of on-chip program memory. The user cannot directly access external memory in the address range of 0 to 2K since this address range is decoded as an internal address. The Z8600 accesses external memory in the following manner:

Table 1. Port 0 Configured to Output A8-A15

<table>
<thead>
<tr>
<th>USER ADDRESS</th>
<th>PHYSICAL MEMORY</th>
<th>LOCATION</th>
<th>DS</th>
<th>ADDRESSES ON PORTS 0 &amp; 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0000-$07FF</td>
<td>NONE</td>
<td>INTERNAL</td>
<td>INACTIVE</td>
<td>0000-07FF</td>
</tr>
<tr>
<td>$0800-$FFFF</td>
<td>$0800-$FFFF</td>
<td>EXTERNAL</td>
<td>ACTIVE</td>
<td>0800-FFFF</td>
</tr>
</tbody>
</table>

NOTE: The external physical addresses $0000-$07FF cannot be accessed.

NOTE: The external physical addresses $0000-$07FF cannot be accessed.
With Port 0 giving the high byte of address and Port 1 giving the low byte of address, a total of 126K bytes of memory can be accessed: 2K bytes of on-chip ROM, 62K bytes of external data memory, and 62K bytes of external program memory.

This scheme does not provide access to the external memory in the address range of 0 to 2K. To access memory in the 0 to 2K range of external memory, the upper address nibble of Port 0 is truncated and address locations 4K to 6K are mapped into the 0 to 2K external memory range as follows:

Table 2. Port 0 Configured to Output A8-A11

<table>
<thead>
<tr>
<th>USER ADDRESS</th>
<th>PHYSICAL MEMORY</th>
<th>LOCATION</th>
<th>DS</th>
<th>ADDRESSES ON PORTS 0 &amp; 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000-07FF</td>
<td>NONE</td>
<td>INTERNAL</td>
<td>INACTIVE</td>
<td>0000-07FF</td>
</tr>
<tr>
<td>0800-0FFF</td>
<td>0800-0FFF</td>
<td>EXTERNAL</td>
<td>ACTIVE</td>
<td>0800-0FFF</td>
</tr>
<tr>
<td>1000-17FF</td>
<td>0000-07FF</td>
<td>EXTERNAL</td>
<td>ACTIVE</td>
<td>0000-07FF</td>
</tr>
</tbody>
</table>

Using the above configuration, memory is accessible in the address range of 0 to 6K. Higher addresses are indistinguishable from the 0 to 6K address space, because the upper four address bits have not been programmed to appear on Port 0.

The Z8600 can access up to 10K of memory using only 12 address lines. It can access 2K of program memory on-chip, 4K of external data memory, and 4K of external program memory for a total of 10K. With only 12 address lines, four lines are released in Port 0 for I/O.

To configure Port 0 as address lines A8-A11 and Port 1 as address/data multiplexed lines AD0-AD7, the following instruction is used:

LD P3M,#(2)XXX10XXX

The following instruction specifies Port 0 as address lines A8-A11 and Port 1 as address/data multiplexed lines AD0-AD7,

LD P01M,#(2)0XX10X1X

The above Xs do not represent "don't care" states. These bits must be set or reset depending on the particular configuration in which the Z8600 is set.

For medium-sized memory applications, the Z8600 can be configured to output address lines A8-A11 on Port 0, address/data multiplexed lines AD0-AD7 on Port 1, and DM on Port 3. In addition, the Z8600 can access a total of 10K bytes of memory.
Z8 Family Software Framing Error Detection

Application Brief

October 1980

INTRODUCTION

The Zilog Z8600 UART microcomputer is a high-performance, single-chip device that incorporates on-chip ROM, RAM, parallel I/O, serial I/O, and a baud rate generator. The UART is capable of full-duplex, asynchronous serial communication at nine standard software-selectable baud rates from 110 to 19.2K baud; other nonstandard rates can also be obtained under software control. Odd parity generation and checking can also be selected.

Three possible error conditions can occur during reception of serial data: framing error, parity error, and overrun error. A framing error condition occurs when a stop bit is not received at the proper time (Figure 1). This can result from noise in the data channel, causing erroneous detection of the previous start bit or lack of detection of a properly transmitted stop bit. The Z8600 UART does not incorporate hardware framing error detection but does facilitate a simple, low-overhead software detection method.

### Fig. 1 - Asynchronous Data Format

<table>
<thead>
<tr>
<th>LSB</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>MSB</th>
</tr>
</thead>
<tbody>
<tr>
<td>START BIT</td>
<td>DATA BITS (8)</td>
<td>PARITY STOP (IF BIT ENABLED)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the middle of the stop bit time, the Z8600 UART automatically posts a serial input interrupt request on IRQ3. The serial input can also be tested by reading Port 3 bit 0 (P30) as shown in Figure 2. Thus, within the interrupt service routine or polling loop, it is only necessary to test P30 in order to identify a framing error. If P30 is Low when IRQ3 goes High, a framing error condition exists and the following code is used to test this:

```
TM P3, #01 ; TEST FOR P30 = 1!
JR Z, FERR ; ELSE FRAMING ERROR!
```

The execution time of this framing error test is only 5.5 µs at 8 MHz. In the worst case (19.2K baud), this would result in 1% overhead. Only five program bytes are required.

### Fig. 2 - Z8600 Serial Input Connection

Z8 is a trademark of Zilog, Inc.
While the Z8600 UART does not incorporate hardware framing error detection, this feature can be implemented in software with a maximum penalty of 1% at 19.2K baud using no additional hardware and only five bytes of program memory.
SECTION 1  

Introduction  
The Z8 is the first microcomputer to offer both a highly integrated microcomputer on a single chip and a fully expandable microprocessor for I/O-and memory-intensive applications. The Z8 has two timer/counters, a UART, 2K bytes internal ROM, and a 144-byte internal register file including 124 bytes of RAM, 32 bits of I/O, and 16 control and status registers. In addition, the Z8 can address up to 124K bytes of external program and data memory, which can provide full, memory-mapped I/O capability.

SECTION 2  

Accessing Register Memory  
The Z8 register space consists of four I/O ports, 16 control and status registers, and 124 general-purpose registers. The general-purpose registers are RAM areas typically used for accumulators, pointers, and stack area. This section describes these registers and how they are used. Bit manipulation and stack operations affecting the register space are discussed in Sections 4 and 5, respectively.

2.1 Registers and Register Pairs. The Z8 supports 8-bit registers and 16-bit register pairs. A register pair consists of an even-numbered register concatenated with the next higher numbered register (%00 and %01, %02 and %03, ... %7E and %7F, %F0 and %F1, ... %FE and %FF). A register pair must be addressed by reference to the even-numbered register. For example, %F1 and %F2 is not a valid register pair; %F0 and %F1 is a valid register pair, addressed by reference to %F0.

Register pairs may be incremented (INCW) and decremented (DECW) and are useful as pointers for accessing program and external data memory. Section 3 discusses the use of register pairs for this purpose.
2.2 Register Pointer. Within the register addressing modes provided by the Z8, a register may be specified by its full 8-bit address (0-%7F, %F0-%FF) or by a short 4-bit address. In the latter case, the register is viewed as one of 16 working registers within a working register group. Such a group must be aligned on a 16-byte boundary and is addressed by Register Pointer RP (%FD). As an example, assume the Register Pointer contains %70, thus pointing to the working register group from %70 to %7F. The LD instruction may be used to initialize register %76 to an immediate value in one of two ways:

LD %76,#1 18-bit register address is given by instruction (3 byte instruction)!

or

LD R6,#1 14-bit working register address is given by instruction; 4-bit working register group address is given by Register Pointer (2 byte instruction)!

The address calculation for the latter case is illustrated in Figure 1. Notice that 4-bit working-register addressing offers code compactness and fast execution compared to its 8-bit counterpart.

To modify the contents of the Register Pointer, the Z8 provides the instruction

SRP #value

Execution of this instruction will load the upper four bits of the Register Pointer; the lower four bits are always set to zero. Although a load instruction such as

LD RP,#value
could be used to perform the same function, SRP provides execution speed (six vs. ten cycles) and code space (two vs. three bytes) advantages over the LD instruction. The instruction

SRP #%70
is used to set the Register Pointer for the above example.

| REGISTER POINTER | 0 1 1 1 0 0 0 0 |
| REGISTER ADDRESS | 0 1 1 1 0 1 1 0 |
| INSTRUCTION (LD RELA) | 0 1 1 0 1 1 0 0 |

Figure 1. Address Calculation Using the Register Pointer

2.3 Context Switching. A typical function performed during an interrupt service routine is context switching. Context switching refers to the saving and subsequent restoring of the program counter, status, and registers of the interrupted task. During an interrupt machine cycle, the Z8 automatically saves the Program Counter and status flags on the stack. It is the responsibility of the interrupt service routine to preserve the register space. The recommended means to this end is to allocate a specific portion of the register file for use by the service routine. The service routine thus preserves the register space of the interrupted task by avoiding modification of registers not allocated as its own. The most efficient scheme with which to implement this function in the Z8 is to allocate a working register group (or portion thereof) to the interrupt service routine. In this way, the preservation of the interrupted task's registers is solely a matter of saving the Register Pointer on entry to the service routine, setting the Register Pointer to its own working register group, and restoring the Register Pointer prior to exiting the service routine. For example, assume such a register allocation scheme has been implemented in which the interrupt service routine for IRQ0 may access only working register Group 4 (registers %40-%4F). The service routine for IRQ0 should be headed by the code sequence:

PUSH RP 1preserve Register Pointer of interrupted task!
SRP #%40 1address working register group 4!

Before exiting, the service routine should execute the instruction

POP RP
to restore the Register Pointer to its entry value.

It should be noted that the technique described above need not be restricted to interrupt service routines. Such a technique might prove efficient for use by a subroutine requiring intermediate registers to produce its outputs. In this way, the calling task can assume that its environment is intact upon return from the subroutine.
2. Accessing Memory (Continued)

2.4 Addressing Mode. The Z8 provides three addressing modes for accessing the register space: Direct Register, Indirect Register, and Indexed.

2.4.1 Direct Register Addressing. This addressing mode is used when the target register address is known at assembly time. Both long (8-bit) register addressing and short (4-bit) working register addressing are supported in this mode. Most instructions supporting this mode provide access to single 8-bit registers. For example:

```
LD %FE,#HI STACK
!load register %FE (SPH) with
  the upper 8-bits of the label
STACK!
AND 0,MASK_REG
!AND register 0 with register
  named MASK_REG!
OR 1,R5 !OR register 1 with working
  register 5!
```

Increment word (INCW) and decrement word (DECW) are the only two Z8 instructions which access 16-bit operands. These instructions are illustrated below for the direct register addressing mode:

```
INCW RR0 !Increment working register
  pair R0, R1:
  R1 ← R1 + 1
  R0 ← R0 + carry!
```
```
DECW %7E
!Decrement working register
  pair %7E, %7F:
%7F ← %7F - 1
%7E ← %7E - carry!
```

Increment word (INCW) and decrement word (DECW) are the only two Z8 instructions which access 16-bit operands. These instructions are illustrated below for the direct register addressing mode:

```
INCW RR0 !Increment working register
  pair R0, R1:
  R1 ← R1 + 1
  R0 ← R0 + carry!
```
```
DECW %7E
!Decrement working register
  pair %7E, %7F:
%7F ← %7F - 1
%7E ← %7E - carry!
```

Note that the instruction

```
INCW RR5
```
will be flagged as an error by the assembler (RR5 not even-numbered).

2.4.2 Indirect Register Addressing. In this addressing mode, the operand is pointed to by the register whose 8-bit register address or 4-bit working register address is given by the instruction. This mode is used when the target register address is not known at assembly time and must be calculated during program execution. For example, assume registers %60-%7F contain a buffer for output to the serial line via repetitive calls to procedure SERIAL_OUT. SERIAL_OUT expects working register 0 to hold the output character. The following instructions illustrate the use of the indirect addressing mode to accomplish this task:

```
LD R1,%20
!working register 1 is the byte
  counter: output %20 bytes!
```
```
LD R2,%60
!working register 2 is the buf-
  fer pointer register!
```
```
out__again:
LD R0,@R2
!load into working register 0
  the byte pointed to by working
  register 2!
INC R2 !increment pointer!
CALL SERIAL_OUT
!output the byte!
DJNZ R1,out__again
!loop till done!
```

Indirect addressing may also be used for accessing a 16-bit register pair via the INCW and DECW instructions. For example,

```
INCW @R0 !Increment the register pair
  whose address is contained in
  working register 0!
```
```
DECW @%7F
!Decrement the register pair
  whose address is contained in
  register %7F!
```

The contents of registers R0 and %7F should be even numbers for proper access; when referencing a register pair, the least significant address bit is forced to the appropriate value by the Z8. However, the register used to point to the register pair need not be an even-numbered register.

Since the indirect addressing mode permits calculation of a target address prior to the desired register access, this mode may be used to simulate other, more complex addressing modes. For example, the instruction

```
SUB 4,BASE(R5)
```
requires the indexed addressing mode which is not directly supported by the Z8 SUBtract instruction. This instruction can be simulated as follows:

```
LD R6,#BASE
!working register 6 has the
  base address!
ADD R6,R5 !calculate the target address!
SUB 4,R6 !now use indirect addressing to
  perform the actual subtract!
```

Any available register or working register may be used in place of R6 in the above example.

2.4.3 Indexed Addressing. The indexed addressing mode is supported by the load instruction (LD) for the transference of bytes between a working register and another register. The effective address of the latter register is given by the instruction which is offset by the contents of a designated working (index)
2. Accessing Register Memory (Continued)

register. This addressing mode provides efficient memory usage when addressing consecutive bytes in a block of register memory, such as a table or a buffer. The working register used as the index in the effective address calculation can serve the additional role of counter for control of a loop's duration.

For example, assume an ASCII character buffer exists in register memory starting at address BUF for LENGTH bytes. In order to determine the logical length of the character string, the buffer should be scanned backward until the first nonoccurrence of a blank character. The following code sequence may be used to accomplish this task:

```
LD R0,#LENGTH    !length of buffer!  
                 !starting at buffer end, look for 1st non-blank!
loop:
  LD R1,BUF-1(R0)  
  CP R1,' '       
  JR ne,found    !found non-blank!  
  DJNZ R0,loop   !look at next!
all_blanks:      !length = 0!
found:
  5 instructions  
  12 bytes        
  1.5 \mu s overhead  
  10.5 \mu s (average) per character tested
```

At labels "all_blanks" and "found," R0 contains the length of the character string. These labels may refer to the same location, but they are shown separately for an application where special processing is required for a string of zero length. To perform this task without indexed addressing would require a code sequence such as:

```
LD R1,#BUF + LENGTH - 1
LD R0,#LENGTH

loop1:        
  CP @R1,' '    
  JR ne,found1 !found non-blank!  
  DEC R1 !dec pointer!  
  DJNZ R0,loop1 !are we done?!  
all_blanks1:  !length = 0!
found1:
  6 instructions  
  13 bytes        
  3 \mu s overhead  
  9.5 \mu s (average) per character tested
```

The latter method requires one more byte of program memory than the former, but is faster by four execution cycles (1 \mu s) per character tested.

As an alternate example, assume a buffer exists as described above, but it is desired to scan this buffer forward for the first occurrence of an ASCII carriage return. The following illustrates the code to do this:

```
LD R0,# - LENGTH     !starting at buffer start, look for 1st carriage return (=\%OD)!
next:
  LD R1,BUF + LENGTH(R0) 
  CP R1,\%OD    !found it!  
  INC R0 !update counter/index!  
  JR nz,next    !try again!  
cr:
  ADD R0,#LENGTH    !R0 has length to CR!
  7 instructions    
  16 bytes          
  1.5 \mu s overhead  
  12 \mu s (average) per character tested
```

SECTION 3

Accessing Program and External Data Memory

In a single instruction, the Z8 can transfer a byte between register memory and either program or external data memory. Load Constant (LDC) and Load Constant and Increment (LDCI) reference program memory; Load External (LDE) and Load External and Increment (LDEI) reference external data memory. These instructions require that a working register pair contain the address of the byte in either program or external data memory to be accessed by the instruction (indirect working register pair addressing mode). The register byte operand is specified by using the direct working register addressing mode in LDC and LDE or the indirect working register addressing mode in LDCI and LDEI. In addition to performing the designated byte transfer, LDCI and LDEI automatically increment both the indirect registers specified by the instruction. These instructions are therefore efficient for performing block moves between register and either program or external data memory. Since the indirect addressing mode is used to specify the operand address within program or external data memory, more complex addressing modes may be simulated as discussed earlier in Section 2.4.2. For example, the instruction

```
LDC R3,BASE(R2)
```

requires the indexed addressing mode, where
3. Accessing Program and External Data Memory (continued)

BASE is the base address of a table in program memory and R2 contains the offset from table start to the desired table entry. The following code sequence simulates this instruction with the use of two additional registers (R0 and R1 in this example).

```
LD R0,#HI BASE
LD R1,#LO BASE
ADD R1,R2
ADC R0,#0
LDC R3,@RRO
```

3.1 Configuring the Z8 for I/O Applications vs. Memory Intensive Applications. The Z8 offers a high degree of flexibility in memory and I/O intensive applications. Thirty-two port bits are provided of which 16, 12, eight, or zero may be configured as address bits to external memory. This allows for addressing of 62K, 4K or 256 bytes of external memory, which can be expanded to 124K, 8K, or 512 bytes if the Data Memory Select output (DM) is used to distinguish between program and data memory accesses. The following instructions illustrate the code sequence required to configure the Z8 with 12 external addressing lines and to enable the Data Memory Select output.

```
LOC OBJ CODE STMT SOURCE STATEMENT
1 SCAN MODULE
2 CONSTANT
3 COUNT := 6
4 GLOBAL
5 $SECTION PROGRAM
6 DELIM ARRAY [COUNT BYTE] := [' ', ';', ':', ',', '.', $0A, $0D]
7 8
P 0000 20 3B 2C
P 0003 2E 0A OD
9 10 PROCEDURE
11 ****************************************************
12 Purpose = To find the next token within an ASCII buffer.
13 Input = RR0 = address of current location within input buffer in external memory.
14 Output = RR4 = address of start of next token
15 RR0 = address of new token's ending delimiter
16 R2 = length of token
17 R3 = ending delimiter
18 R6,R7,R8,R9 destroyed
19 *****************************************************
20 ENTRY
21 clr R2 !init. length counter!
22 DO
23 LDE R3,@RRO !get byte from input buffer!
24 inw RR0 !increment pointer!
25 call check !look for non-delimiter!
26 IF C THEN
27 EXIT !found token start!
28 FI
29 P 0006 B0 E2
30 P 0008 B2 30
31 P 000A A0 E0
32 P 000C D6 002E'
33 P 000F FD 0015'
34 P 0012 8D 0018'
35 P 0015 8D 0008'
```

The two bytes following the mode selection of ports 0 and 1 should not reference external memory due to pipelining of instructions within the Z8. Note that the load instruction to P3M satisfies this requirement (providing that it resides within the internal 2K bytes of memory).

3.2 LDC and LDE. To illustrate the use of the Load Constant (LDC) and Load External (LDE) instructions, assume there exists a hardware configuration with external memory and Data Memory Select enabled. The following module illustrates a program for tokenizing an ASCII input buffer. The program assumes there is a list of delimiters (space, comma, tab, etc.) in program memory at address DELIM for COUNT bytes (accessed via LDC) and that an ASCII input buffer exists in external data memory (accessed via LDE). The program scans the input buffer from the current location and returns the start address of the next token (i.e. the address of the first nondelimiter found) and the length of that token (number of characters from token start to next delimiter).
3. Accessing Program and External Data Memory (Continued)

3.1 ld R4, R0
3.2 ld R5, R1 !RR4 = token starting addr!
3.3 inc R2 !inc. length counter!
3.4 LDE R3, @RR0 !get next input byte!
3.5 call check !look for delimiter!
3.6 IF NC THEN
3.7 EXIT !found token end!
3.8 incw RRO !point to next byte!
3.9 ret
3.10 END scan
3.11

3.2 check PROCEDURE
3.3 !************************************************************************
3.4 Purpose = compare current character with
3.5 delimiter table until table
3.6 end or match found
3.7 !************************************************************************
3.8 input = DELIM = start address of table
3.9 COUNT = length of that table
3.10 R3 = byte to be scrutinized
3.11 output = Carry flag = 1 => input byte
3.12 is not a delimiter (no match found)
3.13 Carry flag = 0 => input byte
3.14 is a delimiter (match found)
3.15 R6, R7, R8, R9 destroyed
3.16 !************************************************************************
3.17 ENTRY
3.18 here:
3.19
3.20 P 002E 6C 00* 36
3.21 P 0030 7C 00* 37
3.22 P 0032 8C 06 38
3.23 P 0034 C2 96 39
3.24 P 0036 A0 E6 40
3.25 P 0038 A2 93 41
3.26 P 003A 6B 03 42
3.27 P 003C 8A F6 43
3.28 P 003E DF 44
3.29 P 003F AF 45
3.30 P 0040 46
3.31
3.32 0 ERRORS
3.33 ASSEMBLY COMPLETE

27 instructions
58 bytes
Execution time is a function of the number of leading delimiters
before token start (x) and the number of characters in the
token (y): 123 μs overhead + 59x μs + 102y μs
(average) per token

3.3 LDCl. A common function performed in Z8 applications is the initialization of the register space. The most obvious approach to this function is the coding of a sequence of “load register with immediate value” instructions (each occupying three program bytes for a register or two program bytes for a working register). This approach is also the most efficient technique for initializing less than eight consecutive registers or 14 consecutive working registers. For a larger register block, the
3. Accessing Program and External Data Memory
(Continued)

LDCI instruction provides an economical means of initializing consecutive registers from an initialization table in program memory. The following code excerpt illustrates this technique of initializing control registers %F2 through %FF from a 14-byte array (INIT_tab) in program memory:

```
SRP #%600
  !RP not %F0!
LD R6,#HI INIT_tab
LD R7,#LO INIT_tab
LD R8,#%F2
  !1st reg to be initialized!
LD R9,#14
  !length of register block!
loop:
  LDCI @R8,@RR6
  !load a register from the init table!
  DINZ R9,loop
  !continue till done!
7 instructions
14 bytes
7.5 µs overhead
7.5 µs per register initialized
```

SECTION 4

Bit Manipulations

Support of the test and modification of an individual bit or group of bits is required by most software applications suited to the Z8 microcomputer. Initializing and modifying the Z8 control registers, polling interrupt requests, manipulating port bits for control or communication with attached devices, and manipulation of software flags for internal control purposes are all examples of the heavy use of bit manipulation functions. These examples illustrate the need for such functions in all areas of the Z8 register space. These functions are supported in the Z8 primarily by six instructions:

- Test under Mask (TM)
- Test Complement under Mask (TCM)
- AND
- OR
- XOR
- Complement (COM)

These instructions may access any Z8 register, regardless of its inherent type (control, I/O, or general purpose), with the exception of the six write-only control registers (PRE0, PRE1, P01M, P2M, P3M, IPR) mentioned earlier in Section 2.1. Table 1 summarizes the function performed on the destination byte by each of the above instructions. All of these instructions, with the exception of COM, require a mask operand. The "selected" bits referenced in Table 1 are those bits in the destination operand for which the corresponding mask bit is a logic 1.

### 3.4 LDEI

The LDEI instruction is useful for moving blocks of data between external and register memory since auto-increment is performed on both indirect registers designated by the instruction. The following code excerpt illustrates a register buffer being saved at address %40 through %60 into external memory at address SAVE:

```
LD R10,#HI SAVE
  !external memory!
LD R11,#LO SAVE
  !address!
LD R8,#%40
  !starting register!
LD R9,#%21
  !number of registers to save in external data memory!
loop:
  LDEI @RR10,@R8
  !load a register!
  DINZ R9,loop
  !until done!
6 instructions
12 bytes
6 µs overhead
7.5 µs per register saved
```

---

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>TM</td>
<td>To test selected bits for logic 0</td>
</tr>
<tr>
<td>TCM</td>
<td>To test selected bits for logic 1</td>
</tr>
<tr>
<td>AND</td>
<td>To reset all but selected bits to logic 0</td>
</tr>
<tr>
<td>OR</td>
<td>To set selected bits to logic 1</td>
</tr>
<tr>
<td>XOR</td>
<td>To complement selected bits</td>
</tr>
<tr>
<td>COM</td>
<td>To complement all bits</td>
</tr>
</tbody>
</table>

**Table 1. Bit Manipulation Instruction Usage**

The instructions AND, OR, XOR, and COM have functions common to today's microprocessors and therefore are not described in depth here. However, examples of the use of these instructions are laced throughout the remainder of this document, thus giving an integrated view of their uses in common functions. Since they are unique to the Z8, the functions of Test under Mask and Test Complement under Mask, are discussed in more detail next.

### 4.1 Test under Mask (TM)

The Test under Mask instruction is used to test selected bits for logic 0. The logical operation performed is destination AND source.

Neither source nor destination operand is modified; the FLAGS control register is the only register affected by this instruction. The zero flag (Z) is set if all selected bits are logic 0; it is reset otherwise. Thus, if the selected destination bits are either all logic 1 or a combination of 1s and 0s, the zero flag would be cleared by this instruction. The sign flag (S) is either set or reset to reflect the result of the
4. Bit Manipulations (Continued)

AND operation; the overflow flag (V) is always reset. All other flags are unaffected. Table 2 illustrates the flag settings which result from the TM instruction on a variety of source and destination operand combinations. Note that a given TM instruction will never result in both the Z and S flags being set.

4.2 Test Complement under Mask. The Test Complement under Mask instruction is used to test selected bits for logic 1. The logical operation performed is (NOT destination) AND source.

<table>
<thead>
<tr>
<th>Destination</th>
<th>Source</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>(binary)</td>
<td>(binary)</td>
<td>Z S V</td>
</tr>
<tr>
<td>10001100</td>
<td>01110000</td>
<td>1 0 0</td>
</tr>
<tr>
<td>01111100</td>
<td>01110000</td>
<td>0 0 0</td>
</tr>
<tr>
<td>10001100</td>
<td>11110000</td>
<td>0 1 0</td>
</tr>
<tr>
<td>11111100</td>
<td>11110000</td>
<td>0 1 0</td>
</tr>
<tr>
<td>00011000</td>
<td>10100001</td>
<td>1 0 0</td>
</tr>
<tr>
<td>01000000</td>
<td>10100001</td>
<td>1 0 0</td>
</tr>
</tbody>
</table>

Table 2. Effects of the TM Instruction

5. Stack Operations

The Z8 stack resides within an area of data memory (internal or external). The current address in the stack is contained in the stack pointer, which decrements as bytes are pushed onto the stack, and increments as bytes are popped from it. The stack pointer occupies two control register bytes (%FE and %FF) in the Z8 register space and may be manipulated like any other register. The stack is useful for subroutine calls, interrupt service routines, and parameter passing and saving. Figure 2 illustrates the downward growth of a stack as bytes are pushed onto it.

5.1 Internal vs. External Stack. The location of the stack in data memory may be selected to be either internal register memory or external data memory. Bit 2 of control register P01M (%F8) controls this selection. Register pair SPH (%FE), SPL (%FF) serves as the stack pointer for an external stack. Register SPL is the stack pointer for an internal stack. In the

As in Test under Mask, the FLAGS control register is the only register affected by this operation. The zero flag (Z) is set if all selected destination bits are 1; it is reset otherwise. The sign flag (S) is set or reset to reflect the result of the AND operation; the overflow flag (V) is always reset. Table 3 illustrates the flag settings which result from the TCM instruction on a variety of source and destination operand combinations. As with the TM instruction, a given TCM instruction will never result in both the Z and S flags being set.

<table>
<thead>
<tr>
<th>Destination</th>
<th>Source</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>(binary)</td>
<td>(binary)</td>
<td>Z S V</td>
</tr>
<tr>
<td>10001100</td>
<td>01110000</td>
<td>0 0 0</td>
</tr>
<tr>
<td>01111100</td>
<td>01110000</td>
<td>1 0 0</td>
</tr>
<tr>
<td>10001100</td>
<td>11110000</td>
<td>0 0 0</td>
</tr>
<tr>
<td>11111100</td>
<td>11110000</td>
<td>1 0 0</td>
</tr>
<tr>
<td>00011000</td>
<td>10100001</td>
<td>0 1 0</td>
</tr>
<tr>
<td>01000000</td>
<td>10100001</td>
<td>0 1 0</td>
</tr>
</tbody>
</table>

Table 3. Effects of the TCM Instruction

5.2 CALL. A subroutine call causes the current Program Counter (the address of the byte following the CALL instruction) to be pushed onto the stack. The Program Counter is loaded with the address specified by the CALL instruction. This address may be a direct address or an indirect register pair reference.

For example,

LABEL 1: CALL %4F98

LABEL 2: CALL @RR4

Figure 2. Growth of a Stack
5. Stack Operations (Continued)

5.3 RET. The return (RET) instruction causes the top two bytes to be popped from the stack and loaded into the Program Counter. Typically, this is the last instruction of a subroutine and thus restores the PC to the address following the CALL to that subroutine.

5.4 Interrupt Machine Cycle. During an interrupt machine cycle, the PC followed by the status flags is pushed onto the stack. (A more detailed discussion of interrupt processing is provided in Section 6.)

5.5 IRET. The interrupt return (IRET) instruction causes the top byte to be popped from the stack and loaded into the status flag register, FLAGS (%FC), the next two bytes are then popped and loaded into the Program Counter. In this way, status is restored and program execution continues where it had left off when the interrupt was recognized.

5.6 PUSH and POP. The PUSH and POP instructions allow the transfer of bytes between the stack and register memory, thus providing program access to the stack for saving and restoring needed values and passing parameters to subroutines.

Execution of a PUSH instruction causes the stack pointer to be decremented by 1; the operand byte is then loaded into the location pointed to by the decremented stack pointer. Execution of a POP instruction causes the byte addressed by the stack pointer to be loaded into the operand byte; the stack pointer is then incremented by 1. In both cases, the operand byte is designated by either a direct register address or an indirect register reference. For example:

PUSH R1 !direct address: push working register 1 onto the stack!

POP 5 !direct address: pop the top stack byte into register 5!

PUSH @R4 !indirect address: pop the top stack byte into the byte pointed to by working register 4!

PUSH @17 !indirect address: push onto the stack the byte pointed to by register 17!

SECTION 6

Interrupts

The Z8 recognizes six different interrupts from four internal and four external sources, including internal timer/counters, serial I/O, and four Port 3 lines. Interrupts may be individual or globally enabled/disabled via Interrupt Mask Register IMR (%FB) and may be prioritized for simultaneous interrupt resolution via Interrupt Priority Register IPR (%FA).

When enabled, interrupt request processing automatically vectors to the designated service routine. When disabled, an interrupt request may be polled to determine when processing is needed.

6.1 Interrupt Initialization. Before the Z8 can recognize interrupts following RESET, some initialization tasks must be performed. The initialization routine should configure the Z8 interrupt requests to be enabled/disabled, as required by the target application and assigned a priority (via IPR) for simultaneous enabled-interrupt resolution. An interrupt request is enabled if the corresponding bit in the IMR is set (= 1) and interrupts are globally enabled (bit 7 of IMR = 1). An interrupt request is disabled if the corresponding bit in the IMR is reset (= 0) or interrupts are globally disabled (bit 7 of IMR = 0).

A RESET of the Z8 causes the contents of the Interrupt Request Register IRQ (%FA) to be held to zero until the execution of an EI instruction. Interrupts that occur while the Z8 is in this initial state will not be recognized, since the corresponding IRQ bit cannot be set. The EI instruction is specially decoded by the Z8 to enable the IRQ; simply setting bit 7 of IMR is therefore not sufficient to enable interrupt processing following RESET. However, subsequent to this initial EI instruction, interrupts may be globally enabled either by the instruction

EI !enable interrupts!

or by a register manipulation instruction such as

OR IMR,#%80

To globally disable interrupts, execute the instruction

DI !disable interrupts!

This will cause bit 7 of IMR to be reset.

Interrupts must be globally disabled prior to any modification of the IMR, IPR or enabled bits of the IRQ (those corresponding to enabled interrupt requests), unless it can be guaranteed that an enabled interrupt will not occur during the processing of such instructions. Since interrupts represent the occurrence of events asynchronous to program execution, it is highly unlikely that such a guarantee can be made reliably.
6. Interrupts
(Continued)

6.2 Vectored Interrupt Processing. Enabled interrupt requests are processed in an automatic vectored mode in which the interrupt service routine address is retrieved from within the first 12 bytes of program memory. When an enabled interrupt request is recognized by the Z8, the Program Counter is pushed onto the stack (low order 8 bits first, then high-order 8 bits) followed by the FLAGS register (%FC). The corresponding interrupt request bit is reset in IRQ, interrupts are globally disabled (bit 7 of IMR is reset), and an indirect jump is taken on the word in location 2x, 2x + 1 (x = interrupt request number, 0 ≤ x ≤ 5). For example, if the bytes at addresses %0004 and %0005 contain %05 and %78 respectively, the interrupt machine cycle for IRQ2 will cause program execution to continue at address %0578.

When interrupts are sampled, more than one interrupt may be pending. The Interrupt Priority Register (IPR) controls the selection of the pending interrupt with highest priority. While this interrupt is being serviced, a higher-priority interrupt may occur. Such interrupts may be allowed service within the current interrupt service routine (nested) or may be held until the current service routine is complete (non-nested).

To allow nested interrupt processing, interrupts must be selectively enabled upon entry to an interrupt service routine. Typically, only higher-priority interrupts would be allowed to nest within the current interrupt service. To do this, an interrupt routine must "know" which interrupts have a higher priority than the current interrupt request. Selection of such nesting priorities is usually a reflection of the priorities established in the Interrupt Priority Register (IPR). Given this data, the first instructions executed in the service routine should be to save the current Interrupt Mask Register, mask off all interrupts of lower and equal priority, and globally enable interrupts (EI). For example, assume that service of interrupt requests 4 and 5 are nested within the service of interrupt request 3. The following illustrates the code required to enable IRQ4 and IRQ5:

```
CONSTANT
  INT__MASK__3 : = %2 00110000

GLOBAL
IRQ3__service PROCEDURE ENTRY
  !service routine for IRQ3!
  PUSH IMR
  !Interrupts were globally disabled during the interrupt machine cycle - no DI is needed prior to modification of IMR!
  AND IMR, #INT__MASK__3
  !disable all but IRQ4 & 5!
  EI
  !...!
  !service interrupt!
  !interrupts are globally enabled now — must disable them prior to modification of IMR!
  DI
  POP IMR
  IRET
END IRQ3__service
```

Note that IRQ4 and IRQ5 are enabled by the above sequence only if their respective IMR bits = 1 on entry to IRQ3__service.

The service routine for an interrupt whose processing is to be completed without interruption should not allow interrupts to be nested within it. Therefore, it need not modify the IMR, since interrupts are disabled automatically during the interrupt machine cycle.

The service routine for an enabled interrupt is typically concluded with an IRET instruction, which restores the FLAGS register and Program Counter from the top of the stack and globally enables interrupts. To return from an interrupt service routine without re-enabling interrupts, the following code sequence could be used:

```
  POP   FLAGS
  !FLAGS ← @SP!
  RET   !PC ← @SP!
```

This accomplishes all the functions of IRET, except that IMR is not affected.

6.3 Polled Interrupt Processing. Disabled interrupt requests may be processed in a polled mode, in which the corresponding bits of the Interrupt Request Register (IRQ) are examined by the software. When an interrupt request bit is found to be a logic 1, the interrupt should be processed by the appropriate
Interrupts
(Continued)

service routine. During such processing, the
interrupt request bit in the IRQ must be
cleared by the software in order for subsequent
interrupts on that line to be distinguished from
the current one. If more than one interrupt
request is to be processed in a polled mode,
polling should occur in the order of estab-
lished priorities. For example, assume that
IRQ0, IRQ1, and IRQ4 are to be polled and
that established priorities are, from high to
low, IRQ4, IRQ0, IRQ1. An instruction
sequence like the following should be used to
poll and service the interrupts:

!poll interrupt inputs here!

<table>
<thead>
<tr>
<th>IRQ4 need service?!</th>
<th>clear IRQ4!</th>
</tr>
</thead>
<tbody>
<tr>
<td>![IRQ4 need service?!]</td>
<td>![clear IRQ4!]</td>
</tr>
<tr>
<td>![IRA0 need service?!]</td>
<td>![clear IRQ0!]</td>
</tr>
<tr>
<td>![IRA1 need service?!]</td>
<td>![clear IRQ1!]</td>
</tr>
<tr>
<td>![IRA4 need service?!]</td>
<td>![clear IRQ4!]</td>
</tr>
<tr>
<td>![IRA0 need service?!]</td>
<td>![clear IRQ0!]</td>
</tr>
<tr>
<td>![IRA1 need service?!]</td>
<td>![clear IRQ1!]</td>
</tr>
</tbody>
</table>

PROCEDURE ENTRY

Each timer/counter is driven by its own 6-bit
prescaler, which is in turn driven by the inter-
nal Z8 clock divided by four. For T1, the inter-
nal clock may be gated or triggered by an
external event or may be replaced by an exter-
nal clock input. Each timer/counter may
operate in either single-pass or continuous
mode where, at end-of-count, either counting
stops or the counter reloads and continues
counting. The counter and prescaler registers
may be altered individually while the timer/
counter is running; the software controls
whether the new values are loaded immedi-
ately or when end-of-count (EOC) is reached.

Although the timer/counter prescaler
registers (PRE0 and PRE1) are write-only,
there is a technique by which the timer/

Timer/Counter Functions
The Z8 provides two 8-bit timer/counters, T0
and T1, which are adaptable to a variety of
application needs and thus allow the software
(and external hardware) to be relieved of the
bulk of such tasks. Included in the set of such
uses are:

- Interval delay timer
- Maintenance of a time-of-day clock
- Watch-dog timer
- External event counting
- Variable pulse train output
- Duration measurement of external event
- Automatic delay following external event
detection

SECTION 7
7. Timer/Counter Functions (Continued) counters may simulate a readable prescaler. This capability is a requirement for high resolution measurement of an event's duration. The basic approach requires that one timer/counter be initialized with the desired counter and prescaler values. The second timer/counter is initialized with a counter equal to the prescaler of the first timer/counter and a prescaler of 1. The second timer/counter must be programmed for continuous mode. With both timer/counters driven by the internal clock and started and stopped simultaneously, they will run synchronous to one another; thus, the value read from the second counter will always be equivalent to the prescaler of the first.

7.1 Time/Count Interval Calculation To determine the time interval \( t \) until EOC, the equation

\[
i = t \times p \times v
\]

characterizes the relation between the prescaler \( p \), counter \( v \), and clock input period \( I \); \( t \) is given by

\[
i = \frac{1}{(\text{XTAL}/8)}
\]

where \( \text{XTAL} \) is the Z8 input clock frequency; \( p \) is in the range \( 1 - 64 \); \( v \) is in the range \( 1 - 256 \). When programming the prescaler and counter registers, the maximum load value is truncated to six and eight bits, respectively, and is therefore programmed as zero. For an input clock frequency of 8 MHz, the prescaler and counter register values may be programmed to time an interval in the range

\[
1 \mu s \times 1 \times 1 \leq i \leq 1 \mu s \times 64 \times 256
\]

\[
1 \mu s \leq i \leq 16.384 \text{ ms}
\]

To determine the count \( c \) until EOC for \( T1 \) with external clock input, the equation

\[
c = p \times v
\]

characterizes the relation between the \( T1 \) prescaler \( p \) and the \( T1 \) counter \( v \). The divide-by-8 on the input frequency is bypassed in this mode. The count range is

\[
1 \times 1 \leq c \leq 64 \times 256
\]

\[
1 \leq c \leq 16.384
\]

7.2 TOUT Modes. Port 3, bit 5 (P36) may be configured as an output (TOUT) which is dynamically controlled by one of the following:

- \( T0 \)
- \( T1 \)
- Internal clock

When driven by \( T0 \) or \( T1 \), TOUT is reset to a logic 1 when the corresponding load bit is set in timer control register TMR (%FI) and toggles on EOC from the corresponding counter. When TOUT is driven by the internal clock, that clock is directly output on P36.

While programmed as TOUT, P36 is disabled from being modified by a write to port register %6; however, its current output may be examined by the Z8 software by a read to port register %6.

7.3 TIN Modes. Port 3, bit 1 (P31) may be configured as an input (TIN) which is used in conjunction with \( T1 \) in one of four modes:

- External clock input
- Gate input for internal clock
- Nonretriggerrable input for internal clock
- Retriggerable input for internal clock

For the latter two modes, it should be noted that the existence of a synchronizing circuit within the Z8 causes a delay of two to three internal clock periods following an external trigger before clocking of the counter actually begins.

Each High-to-Low transition on \( TIN \) will generate interrupt request IRQ2, regardless of the selected \( TIN \) mode or the enabled/disabled state of \( T1 \). IRQ2 must therefore be masked or enabled according to the needs of the application.

The "external clock input" \( TIN \) mode supports the counting of external events, where an event is seen as a High-to-Low transition on \( TIN \). Interrupt request IRQ5 is generated on the \( n \)th occurrence (single-pass mode) or on every \( n \)th occurrence (continuous mode) of that event.

The "gate input for internal clock" \( TIN \) mode provides for duration measurement of an external event. In this mode, the \( T1 \) prescaler is driven by the Z8 internal clock, gated by a High level on \( TIN \). In other words, \( T1 \) will count while \( TIN \) is High and stop counting while \( TIN \) is Low. Interrupt request IRQ2 is generated on the High-to-Low transition on \( TIN \). Interrupt request IRQ5 is generated on \( T1 \) EOC. This mode may be used when the width of a High-going pulse needs to be measured. In this mode, IRQ2 is typically the interrupt request of most importance, since it signals the end of the pulse being measured. If IRQ5 is generated prior to IRQ2 in this mode, the pulse width on \( TIN \) is too large for \( T1 \) to measure in a single pass.

The "nonretriggerrable input" \( TIN \) mode provides for automatic delay timing following an external event. In this mode, \( T1 \) is loaded and clocked by the Z8 internal clock following the first High-to-Low transition on \( TIN \) after \( T1 \) is enabled. \( TIN \) transitions that occur after this point do not affect \( T1 \). In single-pass mode, the
enable bit is reset on EOC; further T\textsubscript{IN} transi-
tions will not cause T\textsubscript{1} to load and begin count-
ing until the software sets the enable bit again. In contin-
uous mode, EOC does not modify the enable bit, but the coun-
ter is reloaded and counting continues immediately; IRQ\textsubscript{5} is
generated every EOC until software resets the enable bit. This T\textsubscript{IN}
mode may be used, for example, to time the line feed delay follow-
ing end of line detection on a printer or to delay data sampling for some length of time following a sample strobe.

The "retriggerable input" T\textsubscript{IN} mode will load and
and clock T\textsubscript{1} with the Z8 internal clock on
every occurrence of a High-to-Low transition on T\textsubscript{IN}. T\textsubscript{1} will time-out and generate interrupt request IRQ\textsubscript{5} when the programmed time
interval (determined by T\textsubscript{1} prescaler and load register values) has elapsed since the last
High-to-Low transition on T\textsubscript{IN}. In single-pass
mode, the enable bit is reset on EOC; further T\textsubscript{IN}
transitions will not cause T\textsubscript{1} to load and
begin counting until the software sets the enable bit again. In continuous mode, EOC does not modify the enable bit, but the coun-
ter is reloaded and counting continues immedi-
ately; IRQ\textsubscript{5} is generated at every EOC until
the software resets the enable bit. This T\textsubscript{IN}
mode may provide such functions as watch-dog
timer (e.g., interrupt if conveyor belt stopped or clock pulse missed), or keyboard time-out
(e.g., interrupt if no input in x ms).

7.4 Examples. Several possible uses of the
timer/counters are given in the following four
eamples.

7.4.1 Time of Day Clock. The followmg
module illustrates the use of T\textsubscript{1} for
maintenance of a time of day clock, which is
kept in binary format in terms of hours,
minutes, seconds, and hundredths of a second. It is desired that the clock be updated once
every hundredth of a second; therefore, T\textsubscript{1} is
programmed in continuous mode to interrupt
100 times a second. Although T\textsubscript{1} is used for
this example, T\textsubscript{0} is equally suited for the task.
The procedure for initializing the timer
(TOD\_INIT), the interrupt service routine
(TOD) which updates the clock, and the in-
terrupt vector for T\textsubscript{1} end-of-count (IRQ\textsubscript{5}) are
illustrated below. XTAL = 7.3728 MHz is
assumed.
7. Timer/Counter Functions (Continued)

7.4.2 Variable Frequency, Variable Pulse Width Output. The following module illustrates one possible use of TOUT. Assume it is necessary to generate a pulse train with a 10% duty cycle, where the output is repetitively high for 1.6 ms and then low for 14.4 ms. To do this, TOUT is controlled by end-of-count from T1, although T0 could alternately be chosen. This example makes use of the Z8 feature that allows a timer's counter register to be modified without disturbing the count in progress. In continuous mode, the new value is loaded when T1 reaches EOC. T1 is first loaded and enabled with values to generate the short interval. The counter register is then immediately modified with the value to generate the long interval; this value is loaded into the counter automatically on T1 EOC. The prescaler selected value must be the same for both long and short intervals. Note that the initial loading of the T1 counter register is followed by setting the T1 load bit of timer control register TMR (%F1); this action causes TOUT to be reset to a logic 1 output. Each subsequent modification of the T1 counter register does not affect the current TOUT level, since the T1 load bit is NOT altered by the software. The new value is loaded on EOC, and TOUT will toggle at that time. The T1 interrupt service routine should simply modify the T1 counter register with the new value, alternating between the long and short interval values.

In the example which follows, bit 0 of register %04 is used as a software flag to indicate which value was loaded last. This module illustrates the procedure for T1/TOUT initialization (PULSE_INIT), the T1 interrupt service routine (PULSE), and the interrupt vector for T1 EOC (IRQ_5). XTAL = 8 MHz is assumed.

```assembly
P 0000 0017' 7 instructions
P 0000 0000 15 bytes
P 0000 0000 16 µs

0 ERRORs
ASSEMBLY COMPLETE

TOD_INIT:
7 instructions
17 bytes
32 bytes
16 µs
19.5 µs (average) including interrupt response time

7.4.2 Variable Frequency, Variable Pulse Width Output. Assume it is necessary to generate a pulse train with a 10% duty cycle, where the output is repetitively high for 1.6 ms and then low for 14.4 ms. To do this, TOUT is controlled by end-of-count from T1, although T0 could alternately be chosen. This example makes use of the Z8 feature that allows a timer's counter register to be modified without disturbing the count in progress. In continuous mode, the new value is loaded when T1 reaches EOC. T1 is first loaded and enabled with values to generate the short interval. The counter register is then immediately modified with the value to generate the long interval; this value is loaded into the counter automatically on T1 EOC. The prescaler selected value must be the same for both long and short intervals. Note that the initial loading of the T1 counter register is followed by setting the T1 load bit of timer control register TMR (%F1); this action causes TOUT to be reset to a logic 1 output. Each subsequent modification of the T1 counter register does not affect the current TOUT level, since the T1 load bit is NOT altered by the software. The new value is loaded on EOC, and TOUT will toggle at that time. The T1 interrupt service routine should simply modify the T1 counter register with the new value, alternating between the long and short interval values.

In the example which follows, bit 0 of register %04 is used as a software flag to indicate which value was loaded last. This module illustrates the procedure for T1/TOUT initialization (PULSE_INIT), the T1 interrupt service routine (PULSE), and the interrupt vector for T1 EOC (IRQ_5). XTAL = 8 MHz is assumed.

Z8ASM 2.0
LOC OBJ CODE STMT SOURCE STATEMENT
1 TIMER2 MODULE
2 $SECTION PROGRAM
3 GLOBAL
4 !IRQ5 interrupt vector!
5 $ABS 10
P 0000 0017' 6 IRQ_5 ARRAY [1 WORD] := [PULSE]
7 $REL
P 000C 8 PULSE_INIT PROCEDURE
10 ENTRY
P 0000 E6 F3 03 11 LD PRE1,#(2)00000011
12 1bit 2-7: prescaler = 64;
13 2bit 1: internal clock;
14 3bit 0: continuous mode
P 0003 E6 F7 00 15 LD P3M,#00
16 4bit 5: let P36 be Tout!
P 0006 E6 F2 19 17 LD T1,#25
18 5for short interval!
P 0009 8F 17 DI
P 000A 46 FB 20 18 OR IMR,#(2)00100000 enable T1 interrupt!
P 000D E6 F1 8C 19 LD TMR,#(2)10001100
20 16bit 6-7: Tout controlled
21 by T1;
22 7bit 3: enable T1;
23 8bit 2: load T1!
P 0010 E6 F2 E1 24 !Set long interval counter, to be loaded on T1 EOC!
P 0010 E6 F2 E1 25 LD T1,#225
26 !Clear alternating flag for PULSE!
```
7. Timer/Counter Functions (Continued)

7.4.3 Cascaded Timer/Counters. For some applications it may be necessary to measure a greater time interval than a single timer/counter can measure (16.384 ms). In this case, TIN and TOUT may be used to cascade T0 and T1 to function as a single unit. TOUT, programmed to toggle on T0 end-of-count, should be wired back to TIN, which is selected as the external clock input for T1. With T0 programmed for continuous mode, TOUT (and therefore TIN) goes through a High-to-Low transition (causing T1 to count) on every other T0 EOC. Interrupt request IRQ5 is generated when the programmed time interval has elapsed. Interrupt requests IRQ2 (generated on every TIN High-to-Low transition) and IRQ4 (generated on T0 EOC) are of no importance in this application and are therefore disabled.

To determine the time interval \( t \) until EOC, the equation

\[
    t = \left( \frac{p0 \times v0 \times (2 \times p1 \times v1 - 1)}{2} \right)
\]

characterizes the relation between the T0 prescaler \( p0 \) and counter \( v0 \), the T1 prescaler \( p1 \) and counter \( v1 \), and the clock input period \( t \). \( t \) is defined in Section 7.1.

Assuming XTAL = 8 MHz, the measurable time interval range is

\[
    1 \mu s \times 1 \times 1 \times (2 \times 1 - 1) \leq t \leq 1 \mu s \times 64 \times 256 \times (2 \times 64 \times 256 - 1)
\]

\[
    1 \mu s \leq t \leq 536.854528 \mu s
\]

Figure 3 illustrates the interconnection between T0 and T1. The following module illustrates the procedure required to initialize the timers for a 1.998 second delay interval:

```
P 0013 B0 04 27 CLR $04
   28 1 = 0 : 25 next;
   29 $04, #1
   30 1 = 1 : 225 next!
   31 END PULSE_INIT
   32
   33 P 0017
   34 PULSE PROCEDURE
   35 ENTRY
   36 P 0017
   37 LD T1,#225
   38 XOR $04, #1
   39 JR Z, PULSE_EXIT
   40 PULSE_EXIT:
   41 IRET
   42 END PULSE
   43 END TIMER2
```

The diagram illustrates the interconnection and the following module indicates the procedure to initialize the timers for a 1.998 second delay interval.

Figure 3. Cascaded Timer/Counters
7. Timer/Counter Functions

7.4.4 Clock Monitor. T1 and TIN may be used to monitor a clock line (in a diskette drive, for example) and generate an interrupt request when a clock pulse is missed. To accomplish this, the clock line to be monitored is wired to P3j (TIN). TIN should be programmed as a retrigerable input to T1, such that each falling edge on TIN will cause T1 to reload and continue counting. If T1 is programmed to time-out after an interval of one-and-a-half times the clock period being monitored, T1 will time-out and generate interrupt request IRQ5 only if a clock pulse is missed.

The following module illustrates the procedure for initializing T1 and TIN (MONITOR_INIT) to monitor a clock with a period of 2 \( \mu \)s. XTAL = 8 MHz is assumed. Note that this example selects single-pass rather than continuous mode for T1. This is to prevent a continuous stream of IRQ5 interrupt requests in the event that the monitored clock fails completely. Rather, the interrupt service routine (CLK_ERR) is left with the choice of whether or not to re-enable the monitoring. Also shown is the T1 interrupt vector (IRQ_5).
I/O Functions

The Z8 provides 32 I/O lines mapped into registers 0-3 of the internal register file. Each nibble of port 0 is individually programmable as input, output, or address/data lines (A[5]-A[12], A[11]-A[8]). Port 1 is programmable as a single entity to provide input, output, or address/data lines (A[7]-A[0]). The operating modes for the bits of Ports 0 and 1 are selected by control register P01M (%F8). Selection of I/O lines as address/data lines supports access to external program and data memory; this is discussed in Section 3. Each bit of Port 2 is individually programmable as an input or an output bit. Port 2 bits programmed as outputs may also be programmed (via bit 0 of P3M) to all have active pull-ups or all be open-drain (active pull-ups inhibited). In Port 3, four bits (P30-P33) are fixed as inputs, and four bits (P34-P37) are fixed as outputs, but their functions are programmable. Special functions provided by Port 3 bits are listed in Table 4. Use of the Data Memory select output is discussed in Section 3; uses of TIN and TOUT are discussed in Section 7.

8.1 Asynchronous Receiver/Transmitter Operation. Full-duplex, serial asynchronous receiver/transmitter operation is provided by the Z8 via P37 (output) and P30 (input) in conjunction with control register SIO (%F0), which is actually two registers: receiver buffer and transmitter buffer. Counter/Timer T0 provides the clock for control of the bit rate.

The Z8 always receives and transmits eight bits between start and stop bits. However, if parity is enabled, the eighth bit (D7) is replaced by the odd-parity bit when transmitted and a parity-error flag (= 1 if error) when received. Table 5 illustrates the state of the parity bit/parity error flag during serial I/O with parity enabled.

Although the Z8 directly supports either odd parity or no parity for serial I/O operation, even parity may also be provided with additional software support. To receive and transmit with even parity, the Z8 should be configured for serial I/O with odd parity disabled. The Z8 software must calculate parity...
and modify the eighth bit prior to the load of a character into SIO and then modify a parity error flag following the load of a character from SIO. All other processing required for serial I/O (e.g., buffer management, error handling, etc.) is the same as that for odd parity operations.

To configure the Z8 for Serial I/O, it is necessary to:

- Enable P3₀ and P3₇ for serial I/O and select parity,
- Set up T₀ for the desired bit rate,
- Configure IRQ3 and IRQ4 for polled or automatic interrupt mode,
- Load and enable T₀.

To enable P3₀ and P3₇ for serial I/O, bit 6 of P3M (R247) is set. To enable odd parity, bit 7 of P3M is set; to disable it, the bit is reset. For example, the instruction

```
LD P3M,#%40
```

will enable serial I/O, but disable parity. The instruction

```
LD P3M,#%C0
```

will enable serial I/O, and enable odd parity.

In the following discussions, bit rate refers to all transmitted bits, including start, stop, and parity (if enabled). The serial bit rate is given by the equation:

\[
\text{bit rate} = \frac{\text{input clock frequency}}{(2 \times 4 \times \text{T₀ prescaler} \times \text{T₀ counter} \times 16)}
\]

The final divide-by-16 is incurred for serial communications, since in this mode T₀ runs at 16 times the bit rate in order to synchronize the data stream. To configure the Z8 for a specific bit rate, appropriate values must first be selected for T₀ prescaler and T₀ counter by the above equation; these values are then programmed into registers T₀ (%F₄) and PRE₀ (%F₅) respectively. Note that PRE₀ also controls the continuous vs. single-pass mode for T₀; continuous mode should be selected for serial I/O. For example, given an input clock frequency of 7.3728 MHz and a selected bit rate of 9600 bits per second, the equation is satisfied by T₀ counter = 2 and prescaler = 3.

The following code sequence will configure the T₀ counter and T₀ prescaler registers:

```
LD T₀,#2 !T₀ counter = 2!
LD PRE₀,#%200001101 !bit 2-7: prescaler = 3; bit 0:
            continuous mode!
```

Interrupt request 3 (IRQ3) is generated whenever a character is transferred into the receive buffer; interrupt request 4 (IRQ4) is generated whenever a character is transferred out of the transmit buffer. Before accepting such interrupt requests, the Interrupt Mask, Request, and Priority Registers (IMR, IRQ, and IPR) must be programmed to configure the mode of interrupt response. The section on Interrupt Processing provides a discussion of interrupt configurations.

To load and enable T₀, set bits 0 and 1 of the timer mode register (TMR) via an instruction such as

```
OR TMR,#%03
```

This will cause the T₀ prescaler and counter registers (PRE₀ and T₀) to be transferred to the T₀ prescaler and counter. In addition, T₀ is enabled to count, and serial I/O operations will commence.

Characters to be output to the serial line should be written to serial I/O register SIO (%F₀). IRQ4 will be generated when all bits have been transferred out.

Characters input from the serial line may be read from SIO. IRQ3 will be generated when a full character has been transferred into SIO.

The following module illustrates the receipt of a character and its immediate echo back to the serial line. It is assumed that the Z8 has been configured for serial I/O as described above, with IRQ3 (receive) enabled to interrupt, and IRQ4 (transmit) configured to be polled. The received character is stored in a circular buffer in register memory from address %42 to %5F. Register %41 contains the address of the next available buffer position and should have been initialized by some earlier routine to %42.

<table>
<thead>
<tr>
<th>Character Loaded Into SIO</th>
<th>Transmitted To Serial Line</th>
<th>Received From Serial Line</th>
<th>Character Transferred To SIO</th>
<th>Note*</th>
</tr>
</thead>
<tbody>
<tr>
<td>01000011</td>
<td>01000011</td>
<td>01000011</td>
<td>01000011</td>
<td>no error</td>
</tr>
<tr>
<td>11000011</td>
<td>01000011</td>
<td>01000011</td>
<td>11000011</td>
<td>error</td>
</tr>
<tr>
<td>01111000</td>
<td>11111000</td>
<td>11111000</td>
<td>01111000</td>
<td>no error</td>
</tr>
<tr>
<td>01111000</td>
<td>11111000</td>
<td>01111000</td>
<td>11111000</td>
<td>error</td>
</tr>
</tbody>
</table>

Table 5. Serial I/O With Odd Parity

* Left-most bit is D7

---

8. I/O Functions
(Continued)
Functions (Continued)

1 SERIAL_IO MODULE
2 CONSTANT
3 next_addr := $41
4 start := $42
5 length := $1E
6 $SECTION PROGRAM
7 GLOBAL
8 !IRQ3 vector!
9 $ABS 6
P 0006 0000' 10 IRQ_3 ARRAY [1 WORD] := [GET_CHARACTER]
11
12 $REL 0
P 0000
13 GET_CHARACTER PROCEDURE ENTRY
14
15 !Serial I/O receive interrupt service!
16 !Echo received character and wait for
17 echo completion!
18
19 P 0000 E4 FO FO
20 !save it in circular buffer!
21 P 0003 F5 F0 41 22 ld %next_addr, SIO !save in buffer!
P 0006 20 41 23 inc next_addr !point to next position!
P 0008 A6 41 60 24 ne next_addr,#start+length
P 000B EB 03 25 jr ne,echo_wait !no.
P 000D E6 41 42 26 ld next_addr,#start+yes. point to start!
27 !now, wait for echo complete!
28 echo_wait:
P 0010 66 FA 29 tcm IRQ,%$10 !transmitted yet!
P 0013 EB FB 30 jr nz,echo_wait !not yet!
P 0015 56 FA EF 31 and IRQ,%$EF !clear IRQ4!
P 0018 BF 32 IRET
P 0019
33 END GET_CHARACTER
35 END SERIAL_IO

10 instructions
25 bytes
35.5 µs + 5.5 µs for each additional pass through the echo_wait loop,
including interrupt response time

8.2 Automatic Bit Rate Detection. In a typical system, where serial communication is
required (e.g. system with a terminal), the
desired bit rate is either user-selectable via a
switch bank or nonvariable and “hard-coded”
in the software. As an alternate method of bit-
rate detection, it is possible to automatically
determine the bit rate of serial data received
by measuring the length of a start bit. The
advantage of this method is that it places no
requirements on the hardware design for this
function and provides a convenient (automatic)
operator interface.

In the technique described here, the serial
channel of the Z8 is initialized to expect a bit
rate of 19,200 bits per second. The number of
bits (n) received through Port pin P30 for each
bit transmitted is expressed by

\[ n = \frac{19,200}{b} \]

where \( b \) = transmission bit rate. For example,
if the transmission bit rate were 1200 bits per
second, each incoming bit would appear to the
receiving serial line as 19,200/1200 or 16 bits.

The following example is capable of disting-
ishing between the bit rates shown in Table 6
and assumes an input clock frequency of
7.3728 MHz, a \( T_0 \) prescaler of 3, and serial I/O
enabled with parity disabled. This example
requires that a character with its low order
bit = 1 (such as a carriage return) be sent to
the serial channel. The start bit of this
character can be measured by counting the
number of zero bits collected before the low
order 1 bit. The number of zero bits actually
collected into data bits by the serial channel is
less than \( n \) (as given in the above equation),
due to the detection of start and stop bits.
Figure 4 illustrates the collection (at 19,200

![Figure 4. Collection of a Start Bit Transmitted at 1.200 BPS and Received at 19.200 BPS](image-url)
Table 6. Inputs to the Automatic Bit Rate Detection Algorithm

<table>
<thead>
<tr>
<th>Bit Rate (bits per second)</th>
<th>Number of 0 Bits Received</th>
<th>Number of Bits Collected as Data Bits</th>
<th>T0 Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>dec - binary</td>
<td>dec - binary</td>
</tr>
<tr>
<td>19200</td>
<td>1</td>
<td>0 - 00000000</td>
<td>1 - 000000000</td>
</tr>
<tr>
<td>9600</td>
<td>2</td>
<td>1 - 00000000</td>
<td>2 - 000000010</td>
</tr>
<tr>
<td>4800</td>
<td>4</td>
<td>3 - 00000011</td>
<td>4 - 000000100</td>
</tr>
<tr>
<td>2400</td>
<td>8</td>
<td>7 - 00000111</td>
<td>8 - 000010000</td>
</tr>
<tr>
<td>1200</td>
<td>16</td>
<td>13 - 0001101</td>
<td>16 - 000100000</td>
</tr>
<tr>
<td>600</td>
<td>32</td>
<td>25 - 0011001</td>
<td>32 - 001000000</td>
</tr>
<tr>
<td>300</td>
<td>64</td>
<td>49 - 0110001</td>
<td>64 - 010000000</td>
</tr>
<tr>
<td>150</td>
<td>128</td>
<td>97 - 1100000</td>
<td>128 - 100000000</td>
</tr>
</tbody>
</table>

Bits per second) of a zero bit transmitted to the Z8 at 1,200 bits per second. Notice that only 13 of the 16 zero bits received are collected as data bits.

Once the number of zero bits in the start bit has been collected and counted, it remains to translate this count into the appropriate T0 counter value and program that value into T0 (%F4). The patterns shown in the two binary columns of Table 6 are utilized in the algorithm for this translation.

As a final step, if incoming data is to commence immediately, it is advisable to wait until the remainder of the current "elongated" character has been received, thus "flushing" the serial line. This can be accomplished either via a software loop, or by programming T1 to generate an interrupt request after the appropriate amount of time has elapsed. Since a character is composed of eight bits plus a minimum of one stop bit following the start bit, the length of time to delay may be expressed as

\[(9 \times n)/b\]

where n and b are as defined above. The following module illustrates a sample program for automatic bit rate detection.
8. I/O Functions (Continued)

P 0036 7B 04 43 jr c, done
P 0038 E0 E2 44 RR R2
data
P 003A 1A F8 45 djnz r1, loop
P 003C 29 F4 46
P 003E D6 0000* 50
call DELAY
P 0041 56 FA F7 52 and IRQ,% F7
P 0044 56 END main
P 0045 54 END bit_rate

0 ERRORS
ASSEMBLY COMPLETE

30 instructions
68 bytes
Execution time is variable based on transmission bit rate.

8.3 Port Handshake. Each of Ports 0, 1 and 2 may be programmed to function under input or output handshake control. Table 7 defines the port bits used for the handshake and the mode bit settings required to select handshaking. To input data under handshake control, the Z8 should read the input port when the DAV input goes Low (signifying that data is available from the attached device). To output data under handshake control, the Z8 should write the output port when the RDY input goes Low (signifying that the previously output data has been accepted by the attached device).

Interrupt requests IRQ0, IRQ1, and IRQ2 are generated by the falling edge of the handshake signal input to the Z8 for Port 0, Port 1, and Port 2 respectively. Port handshake operations may therefore be processed under interrupt control.

Consider a system that requires communication of eight parallel bits of data under handshake from the Z8 to a peripheral device and that Port 2 is selected as the output port. The following assembly code illustrates the proper sequence for initializing Port 2 for output handshake.

<table>
<thead>
<tr>
<th>Port 0</th>
<th>Port 1</th>
<th>Port 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>P32 = DAV</td>
<td>P3 = DAV</td>
<td>P3 = DAV</td>
</tr>
<tr>
<td>P35 = RDY</td>
<td>P34 = RDY</td>
<td>P36 = RDY</td>
</tr>
<tr>
<td>P35 = DAV</td>
<td>P34 = DAV</td>
<td>P36 = DAV</td>
</tr>
</tbody>
</table>

To select input handshake:

{ set bit 6 & reset bit 7 of P0IM (program high nibble as input) }

To select output handshake:

{ reset bits 6, 7 of P0IM (program high nibble as output) }

To enable handshake:

{ set bit 5 of Port 3 (P3g); set bit 2 of P3M }

Table 7. Port Handshake Selection
Arithmetic Routines

This section gives examples of the arithmetic and rotate instructions for use in multiplication, division, conversion, and BCD arithmetic algorithms.

9.1 Binary to Hex ASCII. The following module illustrates the use of the ADD and SWAP arithmetic instructions in the conversion of a 16-bit binary number to its hexadecimal ASCII representation. The 16-bit number is viewed as a string of four nibbles and is processed one nibble at a time from left to right, beginning with the high-order nibble of the lower memory address. %30 is added to each nibble if it is in the range 0 to 9; otherwise %37 is added. In this way, %0 is converted to %30, %1 to %31, ..., %A to %41, ..., %F to %46. Figure 5 illustrates the conversion of RR0 (contents = %F2BE) to its hex ASCII equivalent; the destination buffer is pointed to by RR4.

```
Figure 5. Conversion of (RR0) to Hex ASCII
```

Z8ASM 2.99 INTERNAL RELEASE
LOC OBJ CODE
P 0000
3 BINASC PROCEDURE
4 !*****************************
5 Purpose = To convert a 16-bit binary number to Hex ASCII
6 Input = RR0 = 16-bit binary number.
7 RR4 = pointer to destination buffer in external memory.
8 Output = Resulting ASCII string (4 bytes)
9 in destination buffer.
10 RR4 incremented by 4.
11 R0, R2, R6 destroyed.
12 !*****************************
13 ENTRY
14 again:
15 Id SWAP Id
16 ld R6,#%04 !nibble count!
17 R0, !look at next nibble!
18 P 0004 28 E0 21 ld R2,R0
19 P 0006 56 E2 0F and R2,#%OF !isolate 4 bits!
20 P 0011 06 E2 07 !convert to ASCII: R2 + %30 if R0 in range 0 to 9
21 else R2 + %37 (in range 0A to 0F)
22 !save ASCII in buffer!
23 skipping: Id R4 E4
24 incw RR4 !point to next buffer position!
25 same_byte: 
26 P 0018 A6 06 03 cp R6,#%03 !time for second byte?!
27 P 001B EB E2 02 jr ne,same_byte Inc!
28 P 001D 08 E1 35 Id R0,R1 12nd byte!
29 P 001F 6A E1 37 djnz R6,again
30 P 0021 AF 38 ret
31 P 0022 39 END BINASC
32 40 END ARITH
33 0 errors
34 Assembly complete
35
36 15 instructions
37 34 bytes
38 120.5 µs (average)
39
9. Arithmetic Routines (Continued)

9.2 BCD Addition. The following module illustrates the use of the add with carry (ADC) and decimal adjust (DA) instructions for the addition of two unsigned BCD strings of equal length. Within a BCD string, each nibble represents a decimal digit (0–9). Two such digits are packed per byte with the most significant digit in bits 7–4. Bytes within a BCD string are arranged in memory with the most significant digits stored in the lowest memory location. Figure 6 illustrates the representation of 5970 in a 6-digit BCD string, starting in register %33.

<table>
<thead>
<tr>
<th>BIT</th>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

REGISTER %33 %34 %36

Figure 6. Unsigned BCD Representation

Z8ASM 2.0
LOC OBJ CODE STMT SOURCE STATEMENT

1 ARITH MODULE
2 CONSTANT
3 BCD_SRC := R1
4 BCD_DST := R0
5 BCD_LEN := R2
6 GLOBAL
7 BCDADD PROCEDURE
8 !***************************************************************************
9 Purpose = To add two packed BCD strings of
10 equal length.
11 dst <-- dst + src
12 Input = R0 = pointer to dst BCD string.
13 R1 = pointer to src BCD string.
14 R2 = byte count in BCD string
15 (digit count = (R2)*2 ).
16 Output = BCD string pointed to by R0 is
17 the sum.
18 Carry FLAG = 1 if overflow.
19 R0, R1 as on entry.
20 R2 = 0
21 !***************************************************************************
22 ENTRY
23
24 add BCD_SRC,BCD_LEN !start at least... !
25 P 0000 02 12
26 add BCD_DST,BCD_LEN !significant digits!
27 P 0004 CF
28 ret !carry = 0!
29 add_again:
30 dec BCD_SRC !point to next two
31 P 0005 00 E1
32 dec BCD_DST !point to next two
33 R1 = get src digits!
34 P 0009 E3 31
35 add BCD_DST !src digits!
36 P 000B 13 30
37 ADC R3 !BCD_DST
38 P 000D 40 E3
39 DA R3 !decimal adjust!
40 P 000F F3 03
41 ld BCD_DST,R3 !move to dst!
42 P 0011 2A F2
43 djnz BCD_LEN,add_again !loop for next
44 digits!
45 P 0013 AF
46 ret !all done!
47 P 0014 AF
48 END BCDADD
49 P 0014 AF
50 END ARITH

0 ERRORS
ASSEMBLY COMPLETE

11 instructions
20 bytes
Execution time is a function of the number of bytes (n) in input BCD string:
20 μs + 12.5 (n - 1) μs
9. Arithmetic Routines (Continued)

9.3 Multiply. The following module illustrates an efficient algorithm for the multiplication of two unsigned 8-bit values, resulting in a 16-bit product. The algorithm repetitively shifts the multiplicand right (using RRC), with the low-order bit being shifted out (into the carry flag). If a one is shifted out, the multiplier is added to the high-order byte of the partial product. As the high-order bits of the multiplicand are vacated by the shift, the resulting partial-product bits are rotated in. Thus, the multiplicand and the low byte of the product occupy the same byte, which saves register space, code, and execution time.

```assembly
Z8ASM 2.99 INTERNAL RELEASE
LOC OBJ CODE STMT SOURCE STATEMENT

1 ARITH MODULE
2 CONSTANT
3 MULTIPLIER := R1
4 PRODUCT_LO := R3
5 PRODUCT_HI := R2
6 COUNT := RO
7 GLOBAL

P 0000
8 MULT PROCEDURE
9 !**********************************************************************
10 Purpose = To perform an 8-bit by 8-bit unsigned binary multiplication.
11
12 Input = R1 = multiplier
13 R3 = multiplicand
14
15 Output = RR2 = product
16 RO destroyed
17 !**********************************************************************!
18 ENTRY
P 0000 OC 09
P 0002 B0 E2
P 0004 CF
P 0005 CO E2
P 0007 CO E3
P 0009 FB 02
P 000B 02 21
P 000D 0A F6
P 000F AF
P 0010
20 ld COUNT,09 18 BITS + 1!
21 clr PRODUCT_HI INIT HIGH RESULT BYTE!
22 RCF !CARRY = 0!
23 LOOP: RRC PRODUCT_HI
24 RRC PRODUCT_LO
25 Jr NC,NEXT
26 ADD PRODUCT_HI,MULTIPLIER
27 NEXT: djnz COUNT,LOOP
28 ret
29 END MULT
30 END ARITH

0 errors
Assembly errors complete

9 instructions
16 bytes
92.5 µs (average)
```

9.4 Divide. The following module illustrates an efficient algorithm for the division of a 16-bit unsigned value by an 8-bit unsigned value, resulting in an 8-bit unsigned quotient. The algorithm repetitively shifts the dividend left (via RLC). If the high-order bit shifted out is a one or if the resulting high-order dividend byte is greater than or equal to the divisor, the divisor is subtracted from the high byte of the dividend. As the low-order bits of the dividend are vacated by the shift left, the resulting partial-quotient bits are rotated in. Thus, the quotient and the low byte of the dividend occupy the same byte, which saves register space, code, and execution time.
### 9. Arithmetic Routines

(Continued)

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ CODE</th>
<th>STMT SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>8086</strong></td>
</tr>
<tr>
<td><strong>P 0000</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>P 0002</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>P 0004</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>P 0006</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>P 0008</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>P 000A</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>P 000C</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>P 0010</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>P 0012</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>P 0014</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>P 0015</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>P 0017</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>P 0019</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SECTION 10**

**Conclusion**

This Application Note has focused on ways in which the Z8 microcomputer can easily yet effectively solve various application problems. In particular, the many sample routines illustrated in this document should aid the reader in using the Z8 to greater advantage. The major features of the Z8 have been described so that the user can continue to expand and explore the Z8's repertoire of uses.
Get powerful microprocessor performance by using the Z80. With 158 instructions it offers more flexibility than other µPs, plus 8080 code compatibility.

The Z80 8-bit microprocessor combines all the processing power of the 8080 with 80 additional instructions. And to keep chip count to a minimum, many of the peripheral circuits necessary for 8080 systems have been built into the Z80. All members of the Z80 family are built with n-channel, silicon-gate, depletion-load technology; function at single-phase clock rates of 4 MHz; require just a 5-V supply; and have TTL-compatible inputs and outputs.

The circuit family consists of the Z80-CPU and the following peripherals: a counter-timer circuit (CTC), a parallel input/output circuit (PIO), a direct-memory-access controller (DMA), and a serial input/output circuit (SIO), as well as a group of support boards (Table 1). All the circuits are available in 2.5 or 4-MHz versions, ceramic packages, and extended temperature ranges. All are housed in 40-pin DIPs, except the CTC, which comes in a 28-pin DIP. All peripheral circuits can be daisy-chained for priority interrupt control. Since most peripheral circuits necessary for system operation are built into the Z80, a minimum system consists of the Z80, a system clock, a power-on reset circuit and any memory and peripheral circuits desired (Fig. 1). At the system level, the µP supports vectored priority-interrupt structures without any extra hardware.

Interfaces to the Z80 are simple

Although the Z80 maintains timing and control-signal compatibility with the 8080, it is not pin-compatible. All output lines can sink 1.8 mA at 0.4 V—the equivalent of one standard TTL load.

Three major buses from the chip—the 16-bit address bus, the 8-bit bidirectional data bus and a 13-line control bus—account for 37 of the Z80's 40 pins (Fig. 2). The other three pins are for power, ground and the single-phase clock. Unlike the 8080, the Z80 needs no status latch or clock, and interrupt vectoring and dynamic-memory refresh are completely supported within the µP itself.

The 13 control lines are actually subdivided into three control buses: system control (six lines), µP

<table>
<thead>
<tr>
<th>Table 1. Z80 system components</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part #</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>Z80-CPU</td>
</tr>
<tr>
<td>Z80-CTC</td>
</tr>
<tr>
<td>Z80-PIO</td>
</tr>
<tr>
<td>Z80-DMA</td>
</tr>
<tr>
<td>Z80-SIO</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Support boards</th>
<th>unit qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCB Microcomputer board—kit</td>
<td>$435.</td>
</tr>
<tr>
<td>MDC Memory/floppy-disc controller</td>
<td>$795.</td>
</tr>
<tr>
<td>RMB RAM memory board</td>
<td>$750.</td>
</tr>
<tr>
<td>IOB Input/output board</td>
<td>$350.</td>
</tr>
<tr>
<td>PMB PROM/ROM memory board</td>
<td>$395.</td>
</tr>
<tr>
<td>EPROM EPROM programmer (for 2708)</td>
<td>$475.</td>
</tr>
<tr>
<td>PROM PROM programmer (for 7620, 7640)</td>
<td>$475.</td>
</tr>
<tr>
<td>CPB/ROM Combination programmer</td>
<td>$575.</td>
</tr>
<tr>
<td>VDB Video-display board</td>
<td>$475.</td>
</tr>
</tbody>
</table>

* 4-MHz versions of these parts are available.  ** 0 to 70-C ratings in plastic packages

1. A minimal Z80 system can be built with the µP, an oscillator, some memory and an I/O port such as the PIO. Just a power supply and reset circuit must be added.

Authors: Ralph Ungerman, (former Zilog Vice President); Bernard Peuto (Director of Engineering).

Reprinted with permission of Electronic Design.
2. The three major buses on the Z80 are an address bus, a data bus and a control bus. The control bus can be split into three smaller buses—one for system control, one for processor control and one for bus control.

control (five lines), and µP-bus control (two lines). One bus-control line functions as a bus-request line (BUSRQ), which is an input that requests not only the µP's address and data buses, but also the memory-request, I/O-request, read-data and write-data lines of the system-control bus to go to a high-impedance state so that other devices can use the bus. The other bus-control line, an output signal called bus-acknowledge (BUSAK), goes high to indicate when the lines go into a high-impedance third state.

All six system-control signals are outputs from the µP. An M1 line (machine cycle 1) goes Low to indicate when the µP is in the op-code-fetch part of an instruction. The memory-request line (MREQ) goes Low when the address bus holds a valid address for a memory-read or write operation. An I/O-request line (IORQ) goes Low to indicate that the lower byte of the address bus holds a valid I/O-port address for an I/O-read or write operation.

Memory-read and memory-write lines (RD and WR) are also active when Low. RD indicates that the µP wants to read data from a memory or I/O device, while WR indicates that the data bus holds data to be stored in the addressed location. When the sixth system-control line, a refresh signal (RFSH), goes Low, it indicates that the lower seven bits of the address bus contain a refresh address for dynamic memories, so the current MREQ signal should be used to do a refresh read to all dynamic memory.

The five µP-control lines consist of one output signal and four input lines. All lines are active when Low. The only output is the halt line, which indicates when the µP has executed a software HALT instruction and is waiting for either a nonmaskable or maskable interrupt. While halted, the µP automatically executes NOP instructions to maintain the memory refresh. The wait input (WAIT) indicates to the µP that the addressed memory or I/O device isn't ready for a data transfer (the µP will enter wait states for as long as this line is Low). This line allows memory or peripheral of any speed to be synchronized with the Z80.

To reset the µP or initialize it once it is on, the RESET line can be pulled Low. When pulled Low, it forces the Z80's program counter to 0016, disables the interrupt-enable flip-flop, sets register I to 0016, sets register R to 0016, and sets the interrupt node to 0.

The last two lines are the interrupt-request (INT) and nonmaskable-interrupt (NMI) inputs. When pulled Low, the INT line interrupts the processor at the end of the current instruction if the software-controlled interrupt-enable flip-flop (IFF) is enabled, and if the BUSRQ line is High. Each time the µP accepts an interrupt, an acknowledge signal (IORQ during an MI time) is sent out at the beginning of the next instruction cycle.

The NMI line is a negative-edge triggered input, has a higher priority than the INT line, and is recognized at the end of the current instruction regardless of the IFF state. When triggered, it forces the Z80 to begin execution at location 0066 16 after saving the current contents of the program counter in an external stack.

Interrupts and flags add flexibility

Three interrupt modes are available to the programmer. Mode 0 permits the interrupting device to insert any instruction on the data bus and have the µP execute it. Mode 1 has the µP automatically execute a restart to location 0038 16—no external hardware is required (the contents of the program counter are pushed onto the internal stack).

Mode 2, the most powerful, permits an indirect call
3. By daisy-chaining the peripheral support circuits, any number of peripheral chips can be added to this Z80-based process-control system. The device closest to the μP has to any memory location. In this mode, the μP forms the indirect address from the upper byte of the I register and eight bits that are supplied by the interrupting device.

Two identical 8-bit flag registers (F and F') are part of the Z80. Six of the bits in each register can be used as conditions for jump, call or return instructions; they are set or reset by various μP operations. Both the F and F' registers have four testable flag bits and two nontestable bits. The four testable bits are the Carry flag, Zero flag, Negative-sign flag and Parity/overflow flag.

The Carry flag contains carry from the highest-order accumulator bit—add, subtract, shift and rotate instructions can alter its state. If an operation loads a zero into the accumulator, the Zero flag gets set. Otherwise, it is reset. Used with signed numbers, the Negative-sign flag gets set if the result of an operation is negative (bit 7 of the accumulator is the sign bit). The dual-purpose Parity/overflow bit gets set when the parity of the result in the accumulator for a logic operation is even, or is used to indicate overflow when signed 2's complement arithmetic is performed.

The two nontestable bits are Half-carry and Subtract flags. The Half-carry flag is a BCD-carry or borrow result from the least-significant four bits of the operation. (When a DAA instruction is used, this flag corrects the result of a previously packed decimal-add or subtract operation.) The Subtract flag corrects BCD operations by helping identify the previous instruction; The correction differs for addition and subtraction.

Shifting operations can be performed on any register or memory location rather than just on the accumulator. What's more, I/O operations can also be done with any register, rather than just the accumulator. Sixteen-bit direct loads and stores can be sent to the BC-register pair, the DE pair or the IX or IY registers—instead of just the HL as in the 8080. Consequently, the number of exchange and register-move operations is reduced considerably. Also, 16-bit arithmetic operations using the HL pair
Z80 microprocessor architecture

Built into the Z80 microprocessor are all bus-control, memory-control, and timing signals in addition to eight general-purpose 16-bit registers and an arithmetic-and-logic unit (ALU). The Z80 is upward-compatible with the Intel 8080A and 8085 µPs.

All the 8080 registers are duplicated within the Z80 and, in addition to the eight 8-bit registers (A, F, B, C, D, E, H and L) of the 8080, there is an alternate set (A', F', B', C', D', E', H' and L') and several other special-purpose registers. The additional registers include two 16-bit index registers (IX and IY), an 8-bit interrupt-vector register (I) and an 8-bit memory-refresh register (R). Also carried forward from the 8080 register set are the 16-bit stack pointer and the 16-bit program counter (PC).

Normally, all instructions reference the main register set, and alternate registers are accessed via two exchange commands that swap register contents in the banks. One command, exchanges the accumulator and register flags, while another instruction, exchanges the other six general-purpose registers. Since both instructions are single-byte, minimum-execution-time instructions, a complete swap can be done in four clock cycles (1 μs for a 4-MHz clock). These commands and registers are very handy for rapid single-level interrupt handling.

The Z80's two index registers have no direct corollary in the 8080 architecture, but in operation they resemble the single index register in the 6800 µP. Instructions using this mode such as the accumulator-load command [LD A, (IX + 7)] contain a single-byte offset field (+7, in this case). The effective address of the operand is the sum of the offset and the IX-register contents. This addressing mode is particularly convenient for table references, multibyte entries or for passing a pointer to a group of subroutine parameters. The offset byte is interpreted by the Z80 as a 2's complement number, so both positive and negative indexing is possible.

A special feature of the Z80 is its ability to refresh dynamic memory automatically. Its memory-refresh register acts as a 7-bit counter that is incremented after every op-code fetch. After the fetch, the R-register contents are loaded onto the low-order seven bits of the address bus, and a status line on the processor goes low to indicate the presence of a valid refresh count. Because this entire process takes place while the op code is decoded internally, it never interferes with any other µP activity on the bus.

The I register forms the high-order eight bits of an address. When an interrupt occurs and the Z80 is in the vectored mode, the lower order eight bits are supplied by an interrupting peripheral. In response to the interrupt, the µP does an Indirect Call instruction with the composite address. All the support chips have corresponding registers that store the low-order eight bits and supply them to the Z80 when the interrupt is acknowledged.

Able to perform 12 basic operations—add, subtract, AND, OR, Ex-OR, compare, test-bit, reset-bit, set-bit, increment, decrement, and left or right-shift and rotate (arithmetic or logic)—the ALU communicates with the registers and external-data bus by means of a buffered internal bus. As each instruction is fetched from memory, it is loaded into the instruction register and decoded by the control section, which supplies all the control signals for the Z80's subsystems have been expanded over the 8080's to include add with carry and subtract with borrow.

Software gives the Z80 horsepower

Many of the instructions available only in the Z80 support the manipulation of multibyte blocks of data—a great plus in data communications and text manipulation. For instance, a block-move instruction takes data from the memory location specified by the HL-register pair, deposits them in the location specified by the DE pair, increments the HL and DE registers and then decrements the BC pair, which is assumed to hold a byte counter for the operation. This instruction can be executed in a single cycle or repeat sequence. Decrementing the HL and DE addresses is also possible.

By using the block move command, the µP can transfer bytes of data at 5.25 μs/byte (for a 4-MHz clock). Block operations are also available for memory searches and I/O operations. And shift and rotate operations have been enhanced. For decimal arithmetic, 4-bit shifts through the accumulator can greatly speed up BCD multiplication and division, and bit-manipulation instructions permit fast access to any bit in either the external memory or an internal register.

Other enhancements of the instruction set include
Software capabilities of the Z80

Able to execute over 150 different instructions, including all 78 of the 8080A command set, the Z80 features seven basic families of instructions: load-and-exchange, block-transfer-and-search, arithmetic and logic, bit-manipulation (set, reset and test), jump, call-and-return, input/output, and basic μP-control commands. In all, the Z80 can recognize 696 op codes—244 are the codes of the 8080A.

Load instructions move data internally between μP registers or between the registers and external memory. All these instructions must specify a source location, from which data are to be moved, and a destination location. Block-transfer instructions permit any block of memory to be moved to any other location. Search commands let any block of external memory be examined for any 8-bit character. Once the character is found, the instruction is terminated.

The ALU instructions operate on data held in the accumulator and other general-purpose registers or external memory. Results are held in the accumulator, and appropriate flags are set. Bit-manipulation commands allow any bit in the accumulator, any general-purpose register or any external memory location to be set, reset or tested with a single instruction. Jump, Call and Return instructions are used to transfer between various locations in the program.

I/O instructions permit a wide range of transfers between external memory locations or general-purpose Z80 registers and external I/O devices. In either case, the port number is provided on the lower eight bits of the address bus during any I/O operation. Also, the basic μP-control commands include such instructions as setting or resetting the interruptenable flip-flop or setting the mode of interrupt response.

In addition to the seven addressing modes of the 8080—direct, register, register indirect, modified page 0, extended, implied and immediate—the Z80 has three more addressing modes: relative, indexed, and bit addressing—that can be used.

A special byte-call instruction lets the Z80 program proceed to any of eight locations in page 0 of the memory. This modified page 0 addressing allows a single byte to specify a complete 16-bit address, which saves memory space.

Relative addressing lets the Z80 use the byte following the op code to specify a displacement from the current program-counter value. The displacement value is in 2’s-complement form, which permits up to a +127 or −128 byte displacement. Extended addressing includes two bytes of address in the instruction.

Index registers can also be used as part of the address. In the indexed addressing mode, a byte of data following the op code is a displacement value that must be added to the specified index register (the op code indicates which register) to form a memory pointer. Also available is an implied addressing mode in which the op code uses the contents of one Z80 register or more as the operands. The last addressing mode lets the Z80 access any memory location or μP register and permits any bit to be set, reset or tested.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>8-bit load instructions</strong></td>
<td></td>
</tr>
<tr>
<td>LD r, r'</td>
<td>Load register r with r'</td>
</tr>
<tr>
<td>LD r, n</td>
<td>Load register r with n</td>
</tr>
<tr>
<td>LD r, (HL)</td>
<td>Load r with location (HL)</td>
</tr>
<tr>
<td>LD r, (IX+d)</td>
<td>Load r with location (IX+d)</td>
</tr>
<tr>
<td>LD r, (IY+d)</td>
<td>Load r with location (IY+d)</td>
</tr>
<tr>
<td>LD (HL), r</td>
<td>Load location HL with r</td>
</tr>
<tr>
<td>LD (IX+d), r</td>
<td>Load location IX+d from register r</td>
</tr>
<tr>
<td>LD (IY+d), r</td>
<td>Load location IY+d from register r</td>
</tr>
<tr>
<td>LD (HL), n</td>
<td>Load location HL with value n</td>
</tr>
<tr>
<td>LD (IX+d), n</td>
<td>Load location IX+d with n</td>
</tr>
<tr>
<td>LD (IY+d), n</td>
<td>Load location IY+d with n</td>
</tr>
<tr>
<td>LD A, (BC)</td>
<td>Load AC with location BC</td>
</tr>
<tr>
<td>LD A, (DE)</td>
<td>Load AC with location DE</td>
</tr>
<tr>
<td>LD A, (nn)</td>
<td>Load AC with location nn</td>
</tr>
<tr>
<td>LD (BC), A</td>
<td>Load location BC with AC</td>
</tr>
<tr>
<td>LD (DE), A</td>
<td>Load location DE with AC</td>
</tr>
<tr>
<td>LD (nn), A</td>
<td>Load location nn with AC</td>
</tr>
<tr>
<td>LD A, I</td>
<td>Load register A from I</td>
</tr>
<tr>
<td>LD A, R</td>
<td>Load AC with register R</td>
</tr>
<tr>
<td>LD I, A</td>
<td>Load register I with AC</td>
</tr>
<tr>
<td>LD R, A</td>
<td>Load register R with AC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>16-bit load instructions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LD dd, nn</td>
<td>Load registers dd with nn</td>
</tr>
<tr>
<td>LD IX, nn</td>
<td>Load register IX with nn</td>
</tr>
<tr>
<td>LD IY, nn</td>
<td>Load register IY with nn</td>
</tr>
<tr>
<td>LD HL, (nn)</td>
<td>Load L with contents of location nn and H with (nn+1)</td>
</tr>
<tr>
<td>LD dd, (nn)</td>
<td>Load registers dd with location nn</td>
</tr>
<tr>
<td>LD IX, (nn)</td>
<td>Load IX with location nn</td>
</tr>
<tr>
<td>LD IY, (nn)</td>
<td>Same but for IY</td>
</tr>
<tr>
<td>LD (nn), HL</td>
<td>Load location nn with HL</td>
</tr>
<tr>
<td>LD (nn), dd</td>
<td>Load location (nn) with register pair dd</td>
</tr>
<tr>
<td>LD (nn), IX</td>
<td>Same but for IX</td>
</tr>
<tr>
<td>LD (nn), IY</td>
<td>Same but for IY</td>
</tr>
<tr>
<td>LD SP, HL</td>
<td>Load stack pointer from HL</td>
</tr>
<tr>
<td>LD SP, IX</td>
<td>Load stack pointer from IX</td>
</tr>
<tr>
<td>LD SP, IY</td>
<td>Load stack pointer from IY</td>
</tr>
<tr>
<td>PUSH qq</td>
<td>Load register pair qq onto stack</td>
</tr>
<tr>
<td>PUSH IX</td>
<td>Load IX onto stack</td>
</tr>
<tr>
<td>PUSH IY</td>
<td>Load IY onto stack</td>
</tr>
<tr>
<td>POP qq</td>
<td>Load register pair qq with top of stack</td>
</tr>
<tr>
<td>POP IX</td>
<td>Load IX with top of stack</td>
</tr>
<tr>
<td>POP IY</td>
<td>Load IY with top of stack</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exchange, transfer and search instructions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>EX DE, HL</td>
<td>Exchange contents of DE &amp; HL</td>
</tr>
<tr>
<td>EX AF, A' F'</td>
<td>Exchange contents of AF &amp; A' F'</td>
</tr>
<tr>
<td>EXX</td>
<td>Exchange all six general purpose registers with alternates</td>
</tr>
<tr>
<td>EX (SP), HL</td>
<td>Exchange stack pointer contents with HL</td>
</tr>
<tr>
<td>EX (SP), IX</td>
<td>Same but use IX register</td>
</tr>
<tr>
<td>EX (SP), IY</td>
<td>Same but use IY register</td>
</tr>
<tr>
<td>LDI</td>
<td>Load (HL) into DE, increment DE and HL, decrement BC</td>
</tr>
<tr>
<td>LDIR</td>
<td>Same but loop until (BC) = 0</td>
</tr>
<tr>
<td>LDD</td>
<td>Load location (PE) with location (HL) and decrement DE, HL and BC</td>
</tr>
<tr>
<td>LDTR</td>
<td>Same but loop until (BC) = 0</td>
</tr>
<tr>
<td>CPI</td>
<td>Compare contents of AC with (HL), set Z flat if =, increment HL and decrement BC</td>
</tr>
<tr>
<td>CPRI</td>
<td>Same but repeat until BC = 0</td>
</tr>
<tr>
<td>CP s</td>
<td>Compare operands with AC</td>
</tr>
<tr>
<td>CPD</td>
<td>Same as CPI but decrement HL</td>
</tr>
<tr>
<td>CPDR</td>
<td>Same as CPRI but decrement HL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8-bit arithmetic and logic instructions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD A, r</td>
<td>Add contents of r to AC</td>
</tr>
<tr>
<td>ADD A, n</td>
<td>Add byte n to AC</td>
</tr>
<tr>
<td>ADD A, (HL)</td>
<td>Add contents of HL to AC</td>
</tr>
</tbody>
</table>
### 16-bit Arithmetic Instructions
- **ADD HL, ss**: Add register pair ss to HL
- **ADD HL, ss**: Add with carry operand s to AC
- **SUB s**: Subtract contents of r, n, HL, IX+d or IY+d from AC
- **SBC s**: Same but also subtract carry flag
- **AND s**: Logic AND of operand s and AC
- **OR s**: Same but OR with AC
- **XOR s**: Same but EX-OR with AC
- **INC r**: Increment register r
- **INC (HL)**: Increment location (HL)
- **INC (IX+d)**: Same but use (IX+d)
- **INC (IY+d)**: Same but use (IY+d)
- **DEC m**: Decrement operand m

### Rotate and Shift Instructions
- **RLA**: Rotate AC left
- **RLA**: Same but include carry flag
- **RLA**: Rotate AC right
- **RRA**: Rotate register r left
- **RRC**: Same but include carry flag
- **RRC**: Rotate register r left
- **RLC**: Rotate location (HL) left
- **RLC**: Same but location (IX+d)
- **RLC**: Same but location (IY+d)
- **RL**: Same as any RLC but include carry flag
- **RL**: Same as RLC but shift right
- **RR**: Same as RL m but shift right
- **SLA**: Shift left (any RLC register)
- **SRA**: Same but shift right and keep MSB
- **SRL**: Same as SLA but shift right
- **RLD**: Simultaneous 4-bit rotate from AC₅L to L, H to L and H to AC₅L
- **RRD**: Simultaneous 4-bit rotate from AC₅L to H, L to L and L to AC₅L

### Bit Set, Reset and Test Instructions
- **BIT b, r**: Test bit b of register r
- **BIT b, (HL)**: Test bit b of location (HL)
- **BIT b, (IX+d)**: Test bit b of location (IX+d)
- **BIT b, (IY+d)**: Test bit b of location (IY+d)
- **SET b, r**: Set bit b in register r to 1
- **SET b, (HL)**: Same but use contents of location HL
- **SET b, (IX+d)**: Same but use contents of location IX+d
- **SET b, (IY+d)**: Same but use contents of location IY+d
- **RES b, s**: Reset bit b of operand m

### Jump, Call and Return Instructions
- **JP nn**: Unconditional jump to location nn
- **JP cc, nn**: If condition cc True, do a JP nn otherwise continue
- **JR e**: Unconditional jump to PC+e
- **JR C, e**: If C = 0 continue, if C = 1 do JR e
- **JR NC, e**: Reverse of JR C, e
- **JR Z, e**: If Z = 0 continue, if Z = 1 do JR e
- **JR NZ, e**: Reverse of JR Z, e
- **JP (HL)**: Load PC from (HL)
- **JP (IX)**: Load PC from (IX)
- **JP (IY)**: Load PC from (IY)
- **DJNZ, e**: Decrement register B and jump relative if B ≠ 0

### CALL Instruction
- **CALL cc, nn**: Unconditional call subroutine at location nn
- **CALL cc, nn**: Call subroutine at location nn if condition cc is True

### Return from Subroutine
- **RET**: Return from subroutine
- **RET cc**: If cc false continue, otherwise do RET
- **RETI**: Return from interrupt
- **RETN**: Return from nonmaskable interrupt

### Input/Output Instructions
- **IN A, n**: Load AC with input from device n
- **IN r, (C)**: Load r with input from device C
- **IN r, (C)**: Store contents of location specified by C in address specified by HL, decrement B and increment HL
- **INR**: Same but repeat until B = 0
- **IND**: Same as INI but decrement HL too
- **IND**: Same as INR but decrement HL too
- **OUT n, A**: Load output port (n) with AC
- **OUT n, A**: Load output port (C) with register r
- **OUT**: Load output port (C) with location (HL) and increment HL and decrement B
- **OTIR**: Same but repeat until B = 0
- **OUTD**: Same as OUTI but decrement HL
- **OUTDR**: Same as OTIR but decrement HL

### Notes
- b represents a 3-bit code that indicates position of the bit to be modified
- cc represents a 3-bit code that indicates which of eight condition codes are to be used
- d is an 8-bit offset value
- dd refers to register pairs BC, DC, HL or the stack pointer
- e represents a signed two's complement number between –126 and +129
- m is an 8-bit number
- n is an 8-bit number
- nn refers to two 8-bit bytes
- p represents one of eight restart vector locations on page 0
- pp refers to register pairs BC, DE, the IX register or the stack pointer
- qq refers to register pairs AF, BC, DE or HL
- r or r' refers to registers A, B, C, D, E, H or L or their alternates
- rr refers to register pairs BC, DE, the IX register or the stack pointer
- s refers to either the r registers, the n data word or the contents of locations specified by the contents of the HL, IX+d or IY+d registers
- ss refers to register pairs Bu, u–f, HL or the stack pointer
4. With two parallel, 8-bit I/O ports, the PIO circuit (a) can use either of the ports in a parallel system or on a line-by-line basis for 16 separate I/O lines. Inside each port, five control registers are loaded by the Z80 before operation to initialize the port (b).

the decimal-adjust command, which now works after subtract as well as add operations. Negate-instruction and looping commands are also part of the set. The looping instruction decrements the B register and takes a relative branch if that register has not reached zero. Other operations are shown in the box on Z80 software (see page 58).

Put the Z80 to work

With the four basic Z80 peripheral circuits described virtually any high-performance microcomputer can be constructed. For example, a process-control system can be built around the Z80, as shown in Fig. 3. The peripherals handled by the Z80 controller include three parallel input/output circuits and one counter/timer. The PIOs handle a 16-key keyboard, a printer, a multichannel a/d converter and 16 control lines. Because the peripheral chips can be daisy-chained, a priority interrupt structure with little or no software or hardware overhead. Using the interrupt mode, the requesting PIO causes the \( \mu P \) to go to a service routine, and, after the routine, a special instruction—return-from-interrupt—goes back to the PIO and allows the \( \mu P \) to service lower-priority interrupts.

All support chips have two lines for daisy-chaining—the Interrupt-enable-in (IEI) and Interrupt-enable-out (IEO). Since a CTC is used in the controller to relieve the Z80 from doing timing loops, software overhead is minimized. For the controller of Fig. 3, 14 ICs are needed—and nine of them are memories (2048 bytes of ROM and 4096 bytes of RAM).

The Z80-PIO, a parallel-interface controller, has two 8-bit ports and provides TTL-compatible interfaces (Fig. 4a). Port A has four possible modes of operation: byte output, byte input, byte bidirectional bus and bit. Port B has all the modes except byte bidirectional. The port I/O logic consists of handshake control and six registers (Fig. 4b): an 8-bit input register, an 8-bit output register, a 2-bit mode-control register, an 8-bit mask register, an 8-bit I/O-select register and a 2-bit mask-control register. The last three are used only when the port is programmed to operate in the bit mode. Of the 40 pins on the PIO, 24 are required by the port and CPU buses, six more for \( \mu P \) interfacing, three for interrupt control, four for handshaking the I/O ports and three for power, ground and the single-phase clock.

Four of the six internal registers are loaded by the Z80 for characteristic programming. The contents of the 2-bit mode-control register determine which of the four PIO operating modes is to be used. Similarly, the 2-bit mask-control register specifies the active state (High or Low) of any peripheral-interface lines which are to be monitored. It also permits an interrupt to be generated when all unmasked pins are active (AND condition) or when any unmasked pin is active (OR condition). The code loaded into the mask register determines which peripheral-device interface pins are to be monitored for the specified status condition. And the code held in the I/O-select register determines which pins are inputs or outputs during bit-mode operation. The other two registers hold incoming or outgoing data.

To relieve some software overhead in timing situations, the CTC provides four channels of programmable timing and counting functions that can be set with software (Fig. 5). Each channel operates in either a timer or counter mode, and programmable interrupts can occur on counter or timer states. Other features include a readable down counter, a selectable 16 or 256 clock prescaler for each timer, a selectable positive or negative trigger for timer initiation and automatic reload of counter or timer constants. In addition three channels have zero count/timeout outputs capable of driving Darlington transistors.

Each channel has two registers, both eight bits long and loaded by the \( \mu P \). One register, the time-constant register, loads the preset value into the down counter. The other, called a channel-control register, contains the mode and condition information for channel operation. Also included in each channel are an 8-bit down counter and an 8-bit prescaler. The counter is decremented by the prescaler in the timer mode and by the clock-trigger input in the counter mode.

Of the 28 pins on the CTC, eight connect to the data bus, seven to the control lines, three handle interrupt control and three are required for power, ground and
Each eTe provides four channels of counting/timing capability with an 8-bit counter on each channel (a). There is a control register for each channel and a programmable 8-bit prescaler (b). The single-phase clock. Three of the four input channels have one input and one output line and the fourth channel has only an input line.

**Speed up data transfer with DMA**

One of the interface circuits, a direct-memory-access controller, is designed to effect the high-speed transfer of a block of data between any two ports in a Z80 system and can also be used with other μPs. The circuit is a programmable, single-channel device that provides all address, timing and control signals for the data transfer (Fig. 6). Also, the DMA circuit can search a block of data for a particular, bit-maskable byte, with or without transferring the data. Capable of transfer-only, search-only or search-and-transfer operations at up to 1.2 Mbyte/s, the circuit can automatically increment or decrement the port address from a programmed starting address.

Four communications modes are available on the chip—a byte-at-a-time mode that transfers one byte per request, a burst mode that lets the transfer continue as long as ports are ready, a continuous mode that locks out the μP until the operation is completed, and a transparent mode that steals refresh cycles. When the circuit finds a match or finishes a transfer, it can be programmed to generate an interrupt. Or a complete repeat cycle can be programmed for automatic repeat or repeat on command. A built-in block counter can generate a signal when a certain number of bytes has been transferred—without halting the transfer.

Inside the DMA controller are bus-interface circuits for both the data and address buses, logic and registers to control parameters of the circuit, and address and byte-count circuitry to generate port addresses. There are also provisions for incrementing or decrementing the address, timing circuitry for adjusting the read/write timing of both ports being addressed, and compare logic that permits a byte-matching operation (if a match is encountered, a flag is set in the DMA’s status register). Also built-in is the interrupt and BUSRQ logic, which includes a control register that specifies conditions for the chip to generate an interrupt, all the priority-encoding logic to select between generation of an INT or BUSRQ output, and an interrupt-vector register for automatic vectoring to an interrupt-service routine.

Of the 40 pins on the DMA controller, 24 are needed for the address and data bus, and five are needed for the μP control bus. Eight more handle the interrupt control and timing, and three more are necessary for power, ground and clock inputs.

For serial communications, the serial-input/output circuit (SIO) provides two full duplex programmable channels capable of handling asynchronous, synchronous, and synchronous-bit protocols (IBM Bisync, HDLC and SDLC). It can also generate cyclic-redundancy check codes in any synchronous mode. The SIO has four independent serial ports—two for transmitting and two for receiving (Fig. 7). Asynchronous data with 5, 6, 7 or 8 bits and 1, 1-½ or 2-stop bits as well as even, odd or no-parity generation or checking can be handled.

The circuit has × 1, 16, 32 and 64 clock modes and data rates from 0 to 600 kHz. The transmitter sections have eight modem-control lines, quadruple buffers on receiver data and error registers, and double buffers on the transmitter sections. The bus-I/O control block includes the logic for selecting channels and registers, read/write control, and control of special timing for interrupt-acknowledge cycles. Interrupt logic includes the daisy-chain provision as well as two special 8-bit control registers to handle the various interrupt options, as well as an 8-bit vector register for interrupt response.

Three receive buffers allow enough time for interrupt servicing of fast data rates. The receiver-shift register is controlled by the receive-control logic, which includes two 8-bit registers for receive-mode selection and options. There are two more 8-bit registers for programmable-sync characters. The external-status register is an 8-bit, read-only register that indicates the state of the modem-control pins as well as several internal-status conditions. An internal-status register also indicates the state of the SIO. Each channel has its own receive, transmit and status-register banks.

Now that you are familiar with all the basic system-building blocks, you can mold them with software into
a working system. Because of the Z80's rich instruction set, assembling software programs by hand can be too complicated for most applications; you should use either a dedicated development system or timesharing service.

**Development systems speed software**

The Z80 development systems and the software available from Zilog include several large dedicated units that permit hardware or software development, or both (Table 2). Also available are assemblers, compilers and timesharing services as well as Basic and PLZ. (Cobol and Fortran will be available soon.)

All program statements in the development systems are handled by a text editor and stored in a dual floppy-disc file management system. Once filed, the program is ready for testing and can be translated by an assembler or compiler into code for the Z80. The code can be tested by a hardware/software debug package that provides interrogation, control and tracing capabilities.

In the monitor mode the system has four operating environments: file, edit, debug and assemble. The file capabilities are pretty standard types of features—storing records on disc, pulling records from disc, changing records and saving the new results. The debug and assembler features of the development system offer some pretty powerful capabilities. With the debug commands, you can set up breakpoints, compare blocks of memory and trace an operation.

In the debug mode, for instance, system transactions can be loaded into a special memory as the program executes in real time. And, once any user-defined condition has occurred (such as the setting of bit 6 of port 8B or reading from address 21C8), the program execution can be suspended and the system can re-enter the monitor mode. A complete record of the last 256 transactions just prior to program termination is in the system memory and available to the user.

The main assembler in the development system supports the following features: macros, conditional assembly, the ability to assemble a large file and a sorted-symbol table with cross reference. All these options as well as the printing and listing options are available by setting parameters at the time of assembly. A relocatable assembler with I/O management provides relocatable code and has a linking loader. These permit you also to specify other files that should be included within the current file being operated in asynchronous or synchronous modes, including BiSync and HDLC/SDLC.

---

6. **The direct-memory-access controller** has three classes of operation: transfer-only, search-only or search-and-transfer. Any device on the system bus can be controlled by the DMA; internal counters keep track of source and destination addresses.

7. **Two independent full-duplex serial I/O channels** are built into the SIO. Either channel can be programmed to operate in asynchronous or synchronous modes, including BiSync and HDLC/SDLC.
Table 2. Hardware and software support

<table>
<thead>
<tr>
<th>Type</th>
<th>Price</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Systems</td>
<td></td>
<td>Z80-hardware &amp; software development</td>
<td>3 kbytes ROM, 1 kbyte RAM for system monitor; 16 kbyte RAM; real-time debug module; dual floppy discs; in-circuit emulator; RS-232 or current loop interface; software and user's manuals; extra card slots; 2 chassis system. Universal interface to printers, PROM programmers, etc.</td>
</tr>
<tr>
<td></td>
<td>$8990</td>
<td>system</td>
<td>Same as above, except no in-circuit emulation capability.</td>
</tr>
<tr>
<td></td>
<td>$6990</td>
<td>software development system</td>
<td>Same as first system, except no universal interface.</td>
</tr>
<tr>
<td></td>
<td>$6990</td>
<td>hardware development system</td>
<td>Dual floppy disc system in single chassis containing any combination of Z80 board products (MCB, MDC, etc.)</td>
</tr>
<tr>
<td></td>
<td>$5990</td>
<td>microcomputer system</td>
<td></td>
</tr>
<tr>
<td>Resident software</td>
<td>N.A.</td>
<td>OSZ80-operating system for Z80 development systems and MCB family</td>
<td><strong>Assembler:</strong> translates assembly language mnemonics into machine language. Includes macro's, conditional assembly, the ability to assemble programs of virtually any length and sorted symbol tables with complete cross-reference listings.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Relocating assembler and linking loader:</strong> Facility for linking programs which have been assembled independently and executing.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Editor environment:</strong> allows the user to input and modify texts, such as, assembly language source programs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>File environment:</strong> controls and manipulates disc files that the user creates while writing, debugging and executing programs.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Debug environment:</strong> allows the user to load, test and save programs using an assortment of debugging aids.</td>
</tr>
<tr>
<td></td>
<td>N.A.</td>
<td>BASIC interpreter</td>
<td>This program supports an interpretive language that allows translation into machine code at execution time on a statement-by-statement basis.</td>
</tr>
<tr>
<td></td>
<td>N.A.</td>
<td>PLZ-Zilog resident programming language</td>
<td>From relocatable assembly to high-level system programming:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• allows access to architecture of Z80</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• compiles efficient code</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• easy to translate to machine language</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Two levels of the language allow tailoring to programming task needs.</td>
</tr>
<tr>
<td>Cross software</td>
<td>N.A.</td>
<td>Z80 cross assembler</td>
<td>ANSI 16-Bit Fortran and PLI version available.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Z80-PLM language compiler</td>
<td>Full PLM language compiler produces Z80 code.</td>
</tr>
</tbody>
</table>

assembled so you can combine programs.

The text editor in the system includes many commands (for more than many full minicomputer editors) to help you manipulate the source files. Although it is a line editor (the pointer always indicates the beginning of a line), some string-oriented commands are available. Automatic paging permits you to edit files that are larger than available memory work space. Put and Get commands help you copy sections from one disc file to another or insert them into a program. Over 20 commands in the editor permit text repeats, alterations, storage, line-number printing and macro capabilities.

To develop higher-level language programs, you can use a Basic interpreter. This permits programs to be written and debugged interactively. Also made for resident use is PLZ, a procedure-oriented language with a syntactic and semantic style that blends Algol, PL/I and Pascal. It permits access to the Z80 architecture, can compile efficient code and is easy to translate into machine code. Two levels are available: PLZ Level I combines assembly language with statements necessary to create relocatable program modules; Level II is similar to a high-level systems language in which single statements can substitute for sequences of assembly-language statements.  

2-11
DESIGNING A MICROPROCESSOR DRIVEN MULTIPURPOSE PERIPHERAL CONTROLLER

Requisites of adaptability to mix/match combinations of I/O devices, operation with existing software, and intelligence formulated the design of a microprocessor based multifunction controller architecture.

Richard F. Binder  Modular Computer Systems, Incorporated, Fort Lauderdale, Florida

Requirements for a revised generation of peripheral controllers became apparent while the ModComp CLASSIC computer series was still in the conceptual stage of design. System packaging was based on card-edge pluggable wirewrapped boards for modularity and ease of maintenance. To devote a full board space (approximately 550 integrated circuits) to a single card reader or line printer controller seemed unreasonable; this configuration would waste space and entail extra cost. The decision therefore was made to package several such low performance controllers on one board. Specifying that the design approach would be toward a multiported controller adaptable to many different devices in mix/match configuration avoided the problem of choosing which controllers to conjoin. Also, the new controller had to operate with existing software and would therefore require some intelligence. For example, the existing card reader controller is fully buffered and can transfer data in a direct 12-bit card image; in a transitional 8-bit code called "half-ASCII," packed either one or two bytes per word; or in any 8-bit code downloaded by the host mini-computer, again packed one or two bytes per word. It performs multipunch detection while translating to 8-bit codes. Other controllers to be reimplemented are similarly sophisticated.

Clearly, a microprocessor is the way to package the requisite intelligence on a single board. This approach relieves the designer of complex hardware and/or custom microcode design; a microprocessor's firmware is generally more maintainable than microcode fitted to custom logic. Also, interfacing to future devices should be easier.

General Architecture

Since a microprocessor based controller is extremely slow in relation to a controller implemented with discrete logic, the designer must take into consideration the microprocessor's response time. This response deficiency can be concealed for the most part under the overhead of the host's interrupt-driven input/output (I/O) bus without slowing down the overall system. Several nearly instant system responses are still required, however, such as the setting of controller busy status for the addressed port in response to a transfer-initiate command. These responses are generated by hardware in the form of a programmed logic array (PLA) to set status latches. A Z80A microprocessor computes all other status which are stored as 16-
hit words in four 4-word by 4-bit register files for access by the host's software. Fig 1 is a simplified block diagram of the multifunction controller's final design.

Actual execution of commanded operations is of course carried out by the microprocessor; all commands and data are loaded by the host into the command/data (C/D) first in, first out (FIFO) buffer. This buffer allows the host to issue several commands in rapid sequence. The microprocessor fetches the commands from the buffer one at a time and processes each as required. Even though four independent devices can be controlled by this design, the C/D FIFO buffer need not be very deep in storage capacity; the interrupt-driven I/O bus makes it possible for the microprocessor to control to some extent the rate at which it receives commands by controlling the rate at which it generates interrupts. The C/D FIFO buffer in the controller is 16 words deep by 21 bits wide (16 bits for the data and 5 bits to identify the command's function and destination within the controller).

Similarly, since four independent devices can be controlled, the handling of one device cannot wait for the I/O's response to an interrupt for another device. Therefore, three request FIFO buffers are loaded by the microprocessor for the host: service interrupt (SI), data interrupt (DI), and direct memory processor (DMP). The first two requests are vectored in the host for software processing, while the third activates concurrent hardware in the host's I/O processor itself. As each request is needed, the microprocessor loads the request's source identification word into the appropriate request FIFO buffer. The request FIFO buffers are unloaded by the host at its own rate, and the microprocessor is thereby freed to attend to other functions. Another use of the request FIFO buffers is made by device firmware sets (tasks) which must be able to "stack" more than one request of the same type; a single register for each request type prohibits such stacking.

The microprocessor selected had to be fast enough to support the required system throughput. Tentative short benchmark routines were coded for the 8080A, Z80, 9900, and 6800. One of the coded benchmarks was a routine to fetch the 21-bit contents of the C/D FIFO buffer and transfer control to the appropriate task. The following table gives an approximate comparison of various microprocessors' performance derived from a sample routine based on the controller's firmware.

Fig 1 Controller block diagram. Layout exhibits straightforward bus architecture. Distinguishing feature is addressing scheme consisting of displacement register (DREG) and peripheral-select PLA. This hardware makes possible firmware-transparent bank switching.
Calculations based upon these short routines indicated that of the machines coded for, only the Z80A would be adequate. All further design was tailored explicitly for the Z80A; no detailed hardware or firmware design was produced for the other machines. (These values were attained by a designer most familiar with the Z80. Greater familiarity with other microprocessors might lessen the disparity in performance, but the Z80’s powerful instruction set, vectored interrupt scheme, and twin register sets made it the undisputed choice for this application.)

The four device ports (numbered 0 to 3) must be adaptable to both serial and parallel devices. Originally, the multifunction controller specification called for support of a card reader, three types of line printers (two parallel and one serial), a paper tape punch, a paper tape reader, a serial console terminal, and a full-duplex Rs-232-C asynchronous channel with full modem control and fully programmable parameters. A typical configuration might include a card reader in port 0, a line printer in port 1, and an asynchronous channel in ports 2 and 3. Packaging requirements specified a total of 80 signal pins for all four ports. This constraint, together with an analysis of all the parallel devices, led to a 20-bit port configured as eight bidirectional bits for data transfer, four bidirectional bits for status or control (handshaking, etc), seven input bits for status or control, and one output bit for control (Fig 2).

Of the seven input bits, two can be programmed online for signal inversion, and one of these two can be connected to either a pullup or pulldown resistor for device power sensing. The two groups of bidirectional bits, including control of their buffers, can be reprogrammed online. (For a card reader, all bits are input; for a line printer, all bits are output.) This interface configuration can be made to handle most common 8-bit devices. For serial devices, the 20-pin limitation requires that the parallel buffers be removed and replaced with a universal synchronous/asynchronous receiver/transmitter (USART), as well as appropriate line drivers and receivers.

The Z80A-PIO parallel I/O controller chip provides the required bit-programmable port capability (Fig 2), but it has only two 8-bit ports. Six PIO chips are
needed to drive four 20-bit controller ports. Since one and one-half PIO chips provide 24 bits, the extra four bits control the buffers connected to the programmable bits. The two shared PIO controllers handle only data paths, and therefore are not connected to the microprocessor’s interrupt system. All six chips are configured to operate in bit control mode; hence, their handshake lines are not used. Handshaking is accomplished by addressing various port bits. Each controller port has one complete PIO chip that can generate any needed interrupt.

For serial applications, all 24 bits are available to be programmed as required to best support the specialized serial hardware. To minimize serial hardware, the decision was made to restrict console tasks to port 0 or 1, and the channel task to ports 2 and 3 together. A serial line printer uses the console hardware. A USART is connected so that it is handled as though it were an external device. Serial handling may seem somewhat clumsy, but the hardware involved in the microprocessor’s bus structure is simplified since there is no need to interface directly to a specific chip. This approach also helps to standardize the tasks in their port handling. The Z80-SIO serial I/O chip was not yet available when this controller was designed. Examination of the preliminary SIO specification, however, indicated that use of the SIO would seriously complicate the controller’s internal structure; even if the IC had been available, it probably would not have been used. (The area in question is the displacement register, which will be discussed later.)

Some of the devices to be controlled require either timeouts or cyclic testing of status. These timing functions are triggered by a Z80A-CTC (counter-timer circuit); its four channels are allocated one to each controller port, and are used as timers for intervals up to 16.4 ms (the longest timeout possible with the 4-MHz clock). Longer timeouts are made by firmware counting of CTC interrupts.

A seventh, or frontend, PIO is used between the microprocessor and the host’s I/O to load the various requests into the appropriate FIFO buffers and to provide a vectored interrupt signal to the microprocessor when the C/D FIFO contains information to be processed. Sixteen-bit status and data words for the host are stored in separate 4 x 4 register files whose inputs are I/O mapped for loading by the microprocessor.

**Firmware Considerations**

In order to be able to switch among several concurrent activities, the firmware is designed as a multitasking operating system consisting of an executive program and the various device handlers, or tasks. The executive is always present, while tasks are added as needed by plugging in read-only memory (ROM) sets.

**Executive Program**

The executive occupies 768 bytes of ROM and 256 bytes of random-access memory (RAM), and has three primary functions: to initialize the system, control time-sharing, and provide executive services available to all tasks. System initialization is performed at power-up [Fig 3(a)]. The first routine executed sets up the parameters required for the controller as a whole and initializes the CTC since the latter function is needed only once for all four ports. A loop is then entered which executes four times, once for each port. Task-not-present status is loaded into the status register file, interrupt entry vectors are loaded into the PIO assigned to the port represented by the pass count of the loop (port 0 on the first pass, port 1 on the second, etc), and a test is made to determine whether the port’s task ROM is present. If not, its command entry dedi-
POLLING ROUTINE - TEST EACH TASK SEQUENTIALLY FOR POLLING SERVICE REQUESTS, CALL TASK IF POLL FLAG IS NON-ZERO.

OPEN INTERRUPT WINDOW ONCE EACH PASS.

LD DE,POLF ;FETCH POLL FLAG ADDRESS
LD HL,BADR ;FETCH DISPL REG ADDRESS
POLL EI ;ENABLE INTERRUPTS
INC B
INC B
DISABLE INTERRUPTS FOR POLL SERVICE
LD (HL),B ;WRITE DISPLACEMENT
LT A,(DE) ;THIS PORT NEED POLLING SERVICE?
JR Z,POLL ;NO, TRY NEXT PORT
CALL [CALL POLL ROUTINE INDIRECT]
EXX ;GET OWN REG SET
JP POLL ;NOW GO POLL NEXT PORT
THE Z80 DOES NOT HAVE A CALL-INDIRECT INSTRUCTION -
THE FOLLOWING JUMP SERVES THE PURPOSE.
ICALL JP (HL)

Fig 3  Simplified main controller flow. Controller and four tasks are initialized under control of executive program (a). Program then enters polling loop (b), which provides for priority interrupt service and for one task's round-robin polling service on each pass. In idle condition, loop executes in 11 µs/pass, ensuring reasonably rapid interrupt response.

Once initialized, the system enters an idle loop whose function is to control timesharing among the tasks present. This idle loop, called the polling loop [Fig 3(b)], enables a task in two ways: interrupt service (priority enabling) and polling service (round-robin enabling). Any activity must begin with an interrupt, either from a task's CTC port or from the outside world (the host or the device connected to the particular port). A CTC or device PIO interrupt is vectored to the relevant task routine, which takes appropriate action. An interrupt from the host's I/O, through the frontend PIO, is vectored to an executive routine which extracts the contents of the current C/D
FIFO buffer location, decides whether it is a command or data, and transfers control to the task routine whose address is in the pertinent dedicated location. Whichever task routine is activated completes its action and returns control to the polling loop. The task activity in question may need service of a type which cannot be triggered by further interrupts (such as emptying a buffer asynchronously with its filling, to a device that does not handshake). Such service is activated by the setting of a dedicated location, called the polling flag, to any nonzero value.

Each task has its own polling flag and an associated polling entry dedicated location. During each pass of the polling loop, an interrupt window is opened for 2 μs. If no interrupt is pending, or upon return from the servicing of an interrupt, the loop tests one port's polling flag. If the flag is zero, the port number is incremented and the polling loop restarts, opening the interrupt window. Each port is tested once every four passes through the loop. If the polling flag is nonzero, the loop fetches the address of the task polling routine from the dedicated location and calls that routine. The task routine takes the action for which it has been set up and resets the polling flag if no further polling service is required, and then returns to the polling loop, which continues as before. Note that interrupt service always receives priority over polling service; this arrangement provides the fastest possible response to the outside world, and is guaranteed by specifying that all interrupt routines must enable the interrupt before returning to the polling loop. If another interrupt is pending, it is serviced immediately.

To minimize both interrupt and polling service times, the system takes advantage of the Z80's two sets of working registers. One set contains registers A, B, C, D, E, H, and L; the second set is a duplicate of the first. A single instruction (Exx) will exchange all but A with their duplicates, and another instruction (Ex, Af, Af') will exchange A and the machine's flag register. The latter instruction is not used in the multifunction controller—A is considered volatile by each routine. The polling loop does the context swap for polling routines, but interrupt routines must do the swap themselves. One set is dedicated to the polling loop; register B contains the number of the next port whose polling flag will be tested, register pair DE contains the address of the polling flag in memory, and register pair HL contains the address of the polling routine being called. The second register set is available for use by any task or executive service routine. The Z80 also has two index registers, IX and IY, but these registers are not used in the controller because indexed instructions suffer a 1-μs/instruction time penalty.

The executive provides several services to any task in the form of callable subroutines. These services perform the functions of

1. Decoding commands that a task has determined to be of a control nature, such as controller interrupt connection, data transfer termination, etc. Appropriate action is taken and control is returned to the calling routine if required.

2. Generating one request for a data transfer either to or from the I/O. This request may be either a DI or a DMP request; the executive service routine tests current controller parameters to decide which type is proper.

3. Initializing or terminating the host's DMP hardware by generating specialized DMP requests for these functions.

4. Requesting startup or shutdown service of the host's software by generating an SI, and optionally resetting controller busy when setting the SI.

5. Reinitializing the calling task exactly as is done at power-up. Primarily a diagnostic tool, this function is essentially free—the same routines are used in both cases.

Primary value of the executive services is to reduce the size of the tasks, since each task is limited to 768 bytes of ROM and 256 bytes of RAM. An added advantage lies in the fact that a task designer need not reinvent the wheel by designing all the common functions again for each new task; the effort required to implement new tasks is thereby minimized.

As mentioned above, tasks are limited in size. A more serious problem, however, is the necessity that any task (with certain specific exceptions) be installed into any port position. It is clear that the various port memory areas will have different starting addresses. A conventional software program designed to be loaded into various areas of memory (relocatable software) is accompanied by a list of locations within the program which must be modified upon loading to reflect the program's starting address. Once programmed, however, a ROM set cannot be changed; so it would seem that each task must come in four versions, one for each port. This constraint was considered unacceptable; stocking of all the different ROM sets would create problems for both manufacturer and user. The solution to this problem lies in relocatable firmware, which can be implemented by memory mapping, of which bank switching is a simplified form. Two address bits (A10 and A11) are used to select one of the four tasks, and the most significant address bit bit (A15) is used to control whether the bank switch is invoked [Fig 4(a)]. All tasks, then, can originate at memory address zero. It is possible to address any memory location in absolute mode (A15 = 1), but only the selected task is accessible in relative mode (A15 = 0). The executive is always addressed absolutely to make its services available to any task. The addresses of those services are assembled with each task as "external" equates.

Located in executive RAM, the push-pop stack is addressed absolutely. PIO and CTC interrupt dedicated locations are also in executive RAM, but these locations are addressed relatively so that accesses to the same relative address in each task will be routed to the proper absolute address by the bank switching control hardware. The interrupts themselves are routed through the same absolute addresses by the vectors loaded into the hardware.
**Task Routines**

Tasks consist of a series of short routines whose functions fall into the following categories: initialization, command and data transfer handling, request generation, and device handling. During initialization, the executive passes control to an initializing routine in the task. This initializer is responsible for setting each of its PIOs with the required I/O bit patterns and interrupt enables, and its CTC port with a timeout and an enable if the CTC is to be used. It initializes task oriented, dedicated locations as required, and it generates and loads the proper controller status to the status register file for access by the host. Control is then returned to the executive. This initialization scheme provides the only reasonable means of controller setup—by the tasks themselves.

Commands to a task are received from the C/D FIFO buffer. A FIFO interrupt, recognized by the frontend PIO, triggers the executive routine which fetches the FIFO contents and transfers control to the task's command handler. The command handler then decides what type of action is requested by examining the 16-bit data pattern of the command, making use of executive service if required, and takes that action. Control is then returned to the executive. Note that online task routines must execute as fast as possible in order to make way for other tasks which may be time dependent. In one design case, compliance with this general rule required that an interrupt routine be divided into two portions; the second half of this routine is triggered by programming a PIO to generate an interrupt when an unused output bit is written to the true state. This splitting of a low priority interrupt routine permits higher priority activity to intervene while guaranteeing that the second half will execute much sooner than if it were a polling routine.

At the transfer rates for which the multifunction controller is designed, direct memory access (DMA) adds unnecessary hardware and complicates such capabilities as character recognition and/or processing. Therefore, data transfers are handled in much the same manner as are commands. One of the extra C/D FIFO bits specifies the direction of the transfer; output data from the host are either output directly to the device or loaded into a buffer for output later, when the device is ready. Buffered data output generally is triggered by polling service, whereas direct output always is a result of a transfer requested by a device interrupt routine signifying that the device is ready. Input data may also be buffered or not, as applicable to a particular device; for example, the card reader task buffers its data to protect the I/O against overflow. Input data are loaded into the data register file. When the host accepts the data in response to the controller's data request, that transfer is loaded into the C/D FIFO buffer.
as though it were an output. Upon recognizing this input transfer, the firmware ignores the FIFO data and proceeds to ready the next transfer.

Data requests may be generated by several mechanisms. An interrupt routine servicing a device whose data rate is controlled by the device (e.g., a terminal, through a USART) generates a request when it has data for input or when the device requires output. A polling routine emptying an input buffer generates requests as long as there are data in the buffer. Finally, an output data transfer interrupt routine filling a buffer generates a request every time it is triggered by the receipt of a transfer, after loading the just-received data into its buffer.

Data are transferred to an output device by writing the data to the half PIO and then writing a one followed by a zero to another output bit assigned as the strobe line. If a handshake is required, the strobe is set true and allowed to remain set until an acceptance is signalled by the device. Data from an input device are read from the half PIO and then accepted, if the device requires a response, by strobing in the same manner as for output. The CTC is used for two functions: cyclic activity and single-shot timeouts. Most cyclic activity tests and updates status for devices whose status can change during periods of controller inactivity. Such changes are often due to operator intervention. Single-shot timeouts are required for devices which take long periods to execute some function or functions and do not signal the completion of such functions. A currently supported paper tape punch, for example, takes a full second to run up to speed when started; it is left running for 10 s after the completion of a transfer to avoid repeated up and down cycles.
and the consequent startup delays. Several concurrent timeouts may be controlled by a common clock handler routine, and this activity by no means precludes cyclic functions as well.

**Hardware Architecture**

The memory bank switching function is the central capability of the hardware, and is implemented with a single 2-bit register called the displacement register (DREG). Input to DREG is data bus bits D1 and D2 [Fig 4(a)]. This register is loaded either by an executive routine or hardware interrupt routine. The executive routine which fetches the C/D FIFO contents loads two of the extra FIFO bits into DREG by a mapped memory write. The register is addressed as though it were a memory location; hence, any firmware has the ability to load it, but tasks normally do not do so. The two loaded bits are a binary encode of the port selected by the host’s controller address bus, and when used as A10 and A11, they select the specified task’s memory area. Hardware interrupt response loads D1 and D2 into DREG using the interrupting device’s vector to select the task whose device made the interrupt. Dedicated interrupt entry locations are allocated to provide the proper vectors. It is this function which precluded use of the sio. The sio generates a series of vectors for a given port, so that bits 1 and 2 cannot be used for port selection.

DREG outputs are multiplexed with A10 and A11 from the microprocessor, and the multiplexer is steered by A15. When A15 is a zero (relative mode), the multiplexer gates DREG’s outputs to the controller’s internal address bus, and any one of the four task areas can be accessed. When A15 is a one (absolute mode), the microprocessor’s actual address is used, and any area of memory can be addressed. The executive is always addressed absolutely; certain tasks, which occupy more than one port and are always installed in the same port location, are also addressed absolutely to avoid the necessity of constantly reloading DREG when executing different subroutines.

DREG addresses not only memory but also most other port oriented hardware in the controller. This scheme is necessary to speed execution times; if a task were required to recognize its port address, and compute and load the addresses of all its devices, most routines would become unreasonably long. To avoid this problem, all PIOs and the CTC are selected by a peripheral-select PLA, which is steered by a combination of address bits 0 to 7 and the DREG outputs. DREG steers both data and status register files and most of the port oriented hardware in the front end. This hardware includes a multiplexer whose inputs are the controller’s option-selection switches, and several registers used to control interrupt generation to the host.

In addition, DREG supplies a port selection function in addressing the executive RAM, but in this case DREG’s outputs are multiplexed with address bits A1 and A2. Vectors are loaded into the various ports’ interrupting peripherals, two locations apart, and these two address bits select which port’s dedicated location is addressed when the firmware uses relative mode. For example, the firmware addresses location 4000 (hexadecimal), and any one of the four locations—4000 (equivalent to C000), 4002, 4004, or 4006— is accessed as controlled by DREG [Fig 4(b)]. The firmware cannot address these locations directly in relative mode since DREG overlays the programmed address. During a hardware interrupt response, location C0XX is addressed with the XX being supplied by the interrupting peripheral [Fig 4(c)]. Port 0’s PIO supplies 00 to address C000, port 1’s PIO addresses C002, etc., with A15 forcing absolute addressing to one of four locations which all appear as 4000 to the firmware. This method (Fig 5) is used for all interrupt vectoring. Extended use of DREG makes it unnecessary for a task ever to know in which port it is installed, thereby significantly increasing the overall throughput of the controller.

**Summary**

Although the multifunction controller is limited to an aggregate throughput of from 4000 to 8000 bytes/s, depending upon configuration, this performance exceeds the requirements of the peripheral devices it is designed to handle. The microprocessor based design offers satisfactory solutions to most problems and objectives of a multipurpose intelligent peripheral controller: it allows reasonably fast response to the host, enables the system designer to mix or match peripherals, and provides an adaptable interface for additional peripherals. It can easily be configured for installation into a system, and is relatively inexpensive to manufacture and simple to service.

**Bibliography**


Currently a design engineer and member of the technical staff with MODCOMP, Richard Binder has held various positions in the I/O development group, designing interfaces for an electrostatic printer/plotter, magnetic tape formatters, card reader, moving head discs, and bulk core memory modules. He attended Rose Polytechnic Institute, and has worked as a mechanical designer and technical illustrator.
INTRODUCTION

Interrupts provide a means of processing information on a random or asynchronous basis. The Z80 CPU and peripheral family support interrupts using a daisy-chain approach. As opposed to parallel priority resolution, the daisy chain uses an efficient, minimal-hardware method of prioritizing multiple interrupting devices. In addition, a parallel priority resolution scheme can be configured with the Z80 through the use of a priority encoder and other external hardware.

Coupled with the powerful vectored interrupt capabilities of the Z80, this approach allows the system designer great flexibility in implementing an interrupt driven system.

This document describes the Z80 CPU interrupt process and evaluates the design of the daisy-chain interrupt scheme. The reader can refer to the following documents for additional information:

- Z80 Assembly Language Programming Manual (03-0002-01)
- Z80/Z80A CPU Technical Manual (03-0029-01)
- Z80/Z80A SIO Technical Manual (03-3033-01)
- Z80/Z80A PIO Technical Manual (03-0008-01)
- Microcomputer Components Data Book (03-8032-01)

Figure 1. Z80 Flow Diagram Interrupt Sequence

Non-Maskable Interrupts

The non-maskable interrupt (NMI) is different from the maskable interrupt in several respects. NMI is always enabled and cannot be disabled by the programmer. It is employed when very fast response is desired independent of the maskable interrupt status and can be used for interrupt conditions like a power fail detect. NMI is an edge-sensitive signal that has a lower priority than BUSRQ and higher priority than INT. When the CPU acknowledges an occurrence of NMI, the processor begins a normal opcode fetch. How-
ever, the data read from memory is ignored and instead the CPU restarts its operation from location 66H. The restart operation involves pushing the Program Counter onto the stack, jumping to location 66H, and continuing to process there. During this time, the status of the maskable interrupt condition is preserved and maskable interrupts are disabled, until either an EI instruction is executed or a RETN instruction is used to exit the NMI service routine. The RETN instruction is discussed in detail in the Z80 CPU Technical Manual. Figure 2 shows the timing used for NMI interrupts.

![Figure 2. Non-maskable Interrupt Request Operation](image)

**Maskable Interrupts**

Maskable interrupts (INT) are acknowledged with a lower priority than the NMI but allow the programmer more flexibility. INT is enabled under software control by way of the EI instruction and disabled via the DI instruction. When the Z80 CPU samples INT and it is active, the processor begins an interrupt acknowledge cycle so long as BUSRQ and NMI are not active. The processor does not use an interrupt acknowledge signal but instead issues the acknowledge by executing a special M1 cycle. During an interrupt acknowledge cycle, RD is inactive, IORQ is active, and two wait states are automatically added.

Since the Z80 peripheral devices have logic to interpret this special cycle with no additional external circuitry, a minimal amount of hardware is needed by the system and there is no loss in efficiency. Figure 3 shows the detailed timing for the Z80 CPU interrupt acknowledge cycle.

![Figure 3. Interrupt Acknowledge Cycle](image)
There are also three modes of operation for servicing maskable interrupts. These are Mode 0, Mode 1, and Mode 2. Any particular mode is selected by the programmer using the IM instruction. Figure 4 illustrates the processing sequence for each interrupt mode.

**MODE 0**

- DISABLE INTERRUPTS
  - IFF1, IFF2 = 0
- READ 1ST BYTE OF INSTRUCTION
  - (IIF, IORO LOW)
- MORE BYTES REQUIRED FOR INSTRUCTION
  - NO
  - YES
- READ NEXT BYTE
  - (NORMAL MEM READ WITH PC STATIONARY)
- CALL OR RST
  - YES
  - NO
- EXECUTE INSTRUCTION
  - FOR CALL OR RST
- EI (ENABLE INTERRUPTS)
  - RET STACK → PC

**MODE 1**

- DISABLE INTERRUPTS
  - IFF1, IFF2 = 0
- PC → STACK
- JUMP TO 0038H
- EI (ENABLE INTERRUPTS)
- RET STACK → PC

**MODE 2**

- DISABLE INTERRUPTS
  - IFF1, IFF2 = 0
- READ VECTOR
- PC → STACK
- FORM VECTOR TABLE ADDRESS: IREG + VECTOR
- GET STARTING ADDRESS FROM VECTOR TABLE
- JUMP TO NEW LOCATION
- START INTERRUPT SERVICE ROUTINE
- EI (ENABLE INTERRUPTS)
- RETI
  - STACK → PC

**Figure 4. Maskable Interrupt Sequences**

**Maskable Interrupt Mode 0**

In the maskable interrupt Mode 0 (as with the 8080 interrupt response mode), the interrupting device places an instruction on the data bus for execution by the Z80 CPU. The Instruction used is normally a Restart (RST) Instruction, since this is an efficient one-byte call to any of eight subroutines located in the first 64 bytes of memory. (Each subroutine is a maximum of eight bytes.) However, any Instruction may be given to the Z80 CPU.

The first byte of a multibyte Instruction is read during the interrupt acknowledge cycle. Subsequent bytes are read in by normal memory read cycles. The Program Counter remains at its preinterrupt state, and the user must insure that memory will not respond to these read sequences, since the instruction must come from the interrupt hardware. Timing for the additional bytes of a multibyte Instruction is the same as for a single byte Instruction (see NMI in Figure 2).

When an interrupt is recognized by the CPU, succeeding interrupts are automatically disabled. An EI instruction can be executed anytime after the interrupt sequence begins. The subroutine can then be interrupted, allowing nested interrupts to be used. The nesting process may proceed to any level as long as all pertinent data is saved and restored correctly.

Upon RESET, the CPU automatically sets Interrupt Mode 0.

**Maskable Interrupt Mode 1**

Interrupt Mode 1 provides minimally complex peripherals access to interrupt processing. It is similar to the NMI interrupt, except that the CPU automatically CALLs to location 3BH instead of 66H. As with the NMI, the CPU pushes the Program Counter onto the stack automatically (Figure 2).
The Z80 CPU interrupt vectoring structure allows the peripheral device to identify the starting location of the interrupt service routine.

Mode 2 is the most powerful of the three maskable interrupt modes. It allows an indirect call to any memory location by a single 8-bit vector supplied by the peripheral. In this mode, the peripheral generating the interrupt places the vector onto the data bus in response to an interrupt acknowledge. The vector then becomes the least significant eight bits of the 16-bit indirect pointer, whereas the I register in the CPU forms the most significant eight bits. This address points to an even address in the vector table which then becomes the starting address of the interrupt service routine. Interrupt processing thus starts at an arbitrary 16-bit address, allowing any location in memory to begin the service routine. Since the vector is used to identify two adjacent bytes that form a 16-bit address, the CPU requires an even starting address for the vector's low byte. Figure 5 shows the sequence of events for processing vectored interrupts.

The I register is loaded by the user from the A register. There is no restriction on its value other than its pointing to a valid memory location.

When execution of the interrupt service routine is complete, return to the main program (or another service routine) occurs differently in each mode. In Mode 0, the method of return depends on which instruction was executed by the CPU. If an RST instruction is used, a simple RET suffices. In Mode 1, the CPU treats the interrupt as a CALL instruction, so an RET is used. Mode 2, however, uses the vector information from the peripheral chip to identify the source of the recognized interrupt, and a method of resetting the peripheral's interrupt condition must be found. This is accomplished by using the RETI Instruction. If Mode 2 is used by the programmer, the RETI instruction must be executed in order to utilize the daisy chain properly. Figure 6 shows the RETI instruction timing for the Z80 CPU. A more complete description of how RETI affects the peripherals is given in Chapter 3.

\[\text{Figure 5. Vector Processing Sequence}\]

\[\text{Figure 6. Return From Interrupt Timing (RETI) for Mode 2 Interrupts}\]
Halt Exit Using Interrupts

Whenever a software halt instruction is executed, the CPU enters the Halt state by executing No-OPs (NOPs) until an Interrupt or RESET is received. Each NOP consists of one RT cycle with four T states. The CPU samples the state of the NMI and INT lines on the rising edge of each T4 clock (Figure 7).

When an interrupt exists on either line, the subsequent cycle will be either a memory read operation (NMI) or an interrupt acknowledge (INT). The timing in Figure 7 shows a maskable interrupt causing the CPU to exit the Halt state.

**Figure 7. Exit Halt State with Maskable Interrupt**

Understanding maskable interrupt processing requires a familiarity with how the Z80 peripherals respond to the CPU interrupt sequence. The Z80 family products were designed around the daisy-chain Interrupt configuration, which utilizes minimal external hardware (compared to parallel contention resolution Interrupt priority networks). Many devices handle interrupts via a handshake arrangement, e.g. the use of interrupt request and interrupt acknowledge signals. This is the most straightforward and probably the fastest method of implementing prioritization using more than one interrupting device. However, this method requires a separate interrupt request signal for each peripheral device and either a separate acknowledge signal for each device or a software acknowledge. Extra hardware is needed to provide contention resolution should two or more devices request an interrupt simultaneously. With the Z80 product family, however, such extra hardware is unnecessary and the software does not need to remove the interrupt request from the peripheral device. This is made possible through use of the daisy-chain priority network, which can best be visualized as a type of bucket brigade.

The Z80 peripheral products implement this daisy chain with just three extra signal lines on each chip: interrupt enable input (IEI), interrupt enable output (IEO), and interrupt request (INT). The interrupt request line is an open-drain circuit that is OR wired to the INT pins of the other devices in the chain and connected to the INT pin on the Z80 CPU. This line provides the interrupt request to the CPU.

The IEI and IEO lines provide the means for establishing priority among several requesting devices. The priority of a device is determined by its position in the chain. The IEI pin of the highest priority device in the chain is connected to +5 volts. The IEO pin of the same device is connected to the IEI pin of the next highest priority device. The IEO pin of that device goes to the IEI pin of the next lower device, as shown in Figure 8, and so on to the last device in the chain, where the IEO pin is left open. When a device has an interrupt pending, it activates its INT output which requests service from the CPU and brings its IEO pin Low, thereby preventing the lower devices in the chain from responding to further interrupt operations. When the CPU acknowledges the interrupt, the requesting device removes its interrupt request (INT) signal. After the interrupt processing is completed, the peripheral will reset itself with an RETI instruction, which will bring IEO High and restore the chain to its quiescent state.
NOTES.
1. Device 3 has an interrupt pending (IP set), which causes its IEO pin to go low preventing device 4 from interrupting.
2. CPU acknowledges the interrupt and device 3 has its interrupt under service (IUS set). The device's IP is then reset.
3. Device 1 requests service, suspending device 3 processing. (Assuming interrupts were reenabled.)
4. Device 1 has its interrupt under service.
5. CPU completes processing for device 1 and returns to device 3 service routine.
6. CPU completes processing for device 3 and the daisy chain returns to quiescent state.

Figure 8. 280 Peripheral Device Interrupt Processing Sequence
Interrupt
Acknowledge
Operation

The Z80 peripherals are acknowledged by the CPU and then serviced by an appropriate interrupt service routine. The acknowledge to the peripherals is accomplished by the CPU executing a special MT cycle in which TORQ goes active instead of MREQ and RD. Whenever MT goes active, all peripheral devices are inhibited from changing their interrupt status. This allows time for IEO to propagate through the other devices in the chain before TORQ goes active. As soon as TORQ and MT go active, the peripheral device that has its IEl pin High and an interrupt pending gates an 8-bit vector onto the data bus. (See Figure 9 for timing details.) This 8-bit vector, which was programmed into the peripheral device, is combined with the contents of the I register in the CPU to form a 16-bit address value. During the time that MT and TORQ are active, the requesting device removes the INT signal (since the CPU has acknowledged it) and waits for a return operation. If the peripheral device has its IEl pin High and has had an interrupt acknowledged, then it completes the interrupt cycle and releases IEO (when it sees an RETI instruction [ED-4D sequence] on the data bus). This restores the chain to its normal state so that lower priority interrupts can occur.

The Z80 peripherals monitor MT and RD for the interrupt acknowledge cycle. Since RD goes active before TORQ, the peripheral devices assume an interrupt acknowledge cycle if MT is active and RD is not. This reduces the time required for the internal device logic to respond to TORQ when it goes active.

Thus, a very powerful interrupt-driven system can be implemented with minimal hardware, simple software, and high efficiency using the Z80 family components.

---

Figure 9. Peripheral Interrupt Acknowledge

---

Return from
Interrupt
Operation

When the CPU executes an RETI instruction, the device with an interrupt under service resets its interrupt condition, provided that IEl is High. All Z80 peripheral products sample the data bus for this instruction when MT goes active along with RD.

The RETI instruction decode by the peripheral device has certain characteristics that the designer should be aware of. Since a peripheral can request an interrupt (activate INT and bring IEO Low) at any time, it is possible for a device whose interrupt is currently under service to have its IEl pin Low. This is undesirable, since such a condition prevents the peripheral from resetting IUS properly. To overcome this problem, all Z80 family peripherals bring IEO High momentarily when the ED is seen during the ED-4D instruction fetch. The device whose interrupt is under service does not allow IEO to go High, but when it sees IEl High, it will reset itself when the 4D byte is fetched.

Figure 10 shows the relationship of IP and IUS to INT, IEl, and IEO. IP is set by an interrupt condition on the peripheral (such as the transmit buffer becoming empty) whenever interrupts are enabled. However, IP being set will only cause INT to go active (requesting an interrupt) if IUS is not set and IEl is High. IP is not necessarily cleared by the interrupt acknowledge cycle. Some specific action must be taken within the service routine, such as filling a transmit buffer. Under these conditions, IUS becomes...
set and disables IEO to prevent lower priority devices in the chain from responding to an interrupt cycle. IUS is cleared when IEI is high and the peripheral decodes a valid "ED-4D" Instruction. Thus,

\[ IP = \text{INTACK} \times \text{INT_COND} \]

and

\[ IEO = \text{IEI} \times \text{IUS} \times (IP + "ED") \]

a) State Diagram of 280 Peripherals During Interrupt Cycle

<table>
<thead>
<tr>
<th>IEO</th>
<th>IP</th>
<th>IUS</th>
<th>IEO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

b) Truth Table of Daisy Chain During Idle or Interrupt Acknowledge Condition.

<table>
<thead>
<tr>
<th>IEO</th>
<th>IP</th>
<th>IUS</th>
<th>IEO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 10. 280 Peripheral Interrupt States

DAISY CHAIN DESIGN CONSIDERATIONS

There are several aspects of the Z80 family daisy chain implementation that deserve further attention.

First, since the peripheral devices must be able to monitor the data bus in order to decode the RETI instruction properly, a means of allowing them access to the data bus must be provided if buffers are used. This can be done by simply enabling the buffers from the data bus to the peripheral for all conditions except I/O read and interrupt acknowledge. Since the peripheral must assert an 8-bit vector during interrupt acknowledge, the buffers must also accommodate this.

Second, because the peripheral devices have a finite time during which IEI and IEO can stabilize within, the propagation delay of the devices must be taken into consideration. Since a device can change its Interrupt status until reaching the active edge of MI during interrupt acknowledge, the time from this edge until IORQ becomes active is the time in which the daisy chain must stabilize. Figure 11 shows the timing relationships involved in this process.

**Figure 11. Interrupt Acknowledge Peripheral Propagation Delay**
The Z80 CPU automatically inserts two wait states during INTACK, allowing a worst-case time for a chain of four devices to become settled (when using Z80A CPU and peripherals at 4MHz). If more devices are in the chain, some other means of stabilizing the chain must be provided. This can be done either by adding additional wait states to the INTACK cycle or by providing logic to the peripherals that allows faster propagation time down the chain. Figure 12 shows circuitry that provides both additional wait states and an interrupt look-ahead circuit when more than four peripheral devices are connected to the daisy chain.

When adding wait states to the Z80 CPU interrupt acknowledge cycle, care must be taken to insure that IORQ goes active at the proper time. Normally, the CPU activates IORQ on the falling edge of the clock during the first wait cycle. If external logic is used to insert additional wait states, these are appended to the two wait states already generated by the CPU. Because IORQ goes active during the first wait state and the peripherals assert their vectors when IORQ becomes active, IORQ must be inhibited until the daisy chain becomes stable. This can be done simply by adding a few gates to the wait logic (Figure 13). IORQ' is the delayed IORQ that activates the peripheral devices.

Figure 12A. Daisy Chain Look-Ahead Logic for More Than Four Peripheral Devices
The propagation delay through the peripheral devices applies during the return from interrupt condition, also. Worst-case timing involves the lowest priority device that has an interrupt under service and the highest priority device that has an interrupt pending. When the ED part of the RETI opcode is fetched, the peripheral devices must decode it, and the highest priority device must bring its IEO pin High. This IEO high signal must then propagate through the chain down to the lowest priority device before the 4D part of RETI is decoded. Figure 14 shows the timing relationships involved. This timing is not as critical as the interrupt acknowledge timing at 4 MHz, but should be considered if wait states are being added to the INTCycle.

If using nested interrupts with a large daisy chain, the programmer should be careful not to place the RETI opcodes too close together. Since RETI is 14 cycles long, this is generally not a problem unless a very long chain is used.

**Figure 13. Wait State Logic for Interrupt Acknowledge Cycle.**
Counter Preset Value Should Be S-n. Where n = # Wait State Added

**Figure 14. Daisy Chain Interrupt Timing (RETI Condition)**

**Notes:**
1. Setup time for IEO to "4D" decode = 200ns (4 MHz)
2. Must look at IEO during ED-4D because nested interrupts allow more than 1 IUS latch to be set at one time.
3. Delay time from ED decode with IP set to IEO high
   = 300ns (typ) 400ns (max) @2.5 MHz. This in addition to ripple time for other devices in chain.

$T_p = T_{d,ED(IEO)} + T_{d,IEI(IEO) \cdot [N-2]} + T_{d,IEI(4D)}$

For N-2 devices

$T_{d,ED(IEO)} =$ Delay time from "ED" decode to IEO rise.

$T_{d,IEI(IEO)} =$ Delay time from IEO high to IEO rise.

$T_{d,IEI(4D)} =$ Setup time for IEO during "4D" decode.

(For last device in chain.)
Interfacing Zilog 8500 series peripheral products (CIO, FIO, SCC, etc.) to the Z80 CPU is a little different from interfacing the Z80 peripherals to the CPU. The primary difference between the Z80-type peripherals and the 8500-type peripherals is in the interrupt acknowledge circuitry. Functionally, they are the same, as can be seen in the timing diagrams of Figure 15. However, the 8500 peripherals do not sample MI, RD, and IORQ for the interrupt acknowledge, but have an explicit INTACK pin to signal the interrupt acknowledge. Also, since the 8500 peripherals have a software reset for the interrupt under service flip-flop, these devices do not require a special return opcode to do that operation. The user need only be concerned with the interrupt acknowledge timing when using the 8500-type peripherals.

Figure 16 shows a circuit that provides wait states for the Z80 CPU interrupt acknowledge cycle in addition to INTACK generation. The TORQ circuitry can be omitted if no Z80 family peripheral devices are used.

In each case, the 8500 peripheral component requires INTACK and RD to be active in order for the interrupt vector to be made available to the CPU. The logic shown provides for this.

This circuitry also permits extended interrupt acknowledge times to allow for the daisy chain propagation delay and the vector response delay, so that larger chains can be implemented.

**Figure 15. Timing for 8500 Peripherals During Interrupt Acknowledge.**

**Figure 16. Interface Logic For Connecting 8500 Series Peripherals To Z80 System.**

**NOTE:**
1. RD and WR should only be connected to 8500 peripherals and not to Z80 peripherals.
A RESET to the Z80 CPU does several things as far as interrupts are concerned. The I register, which contains the upper eight bits of the 16-bit interrupt address value, is reset to 0, and the interrupt mode is set to Mode 0. Maskable interrupts are disabled until the programmer instructs the CPU to execute an EI instruction, just as if a DI instruction were executed. If an NMi occurs during the RESET operation, the CPU executes one instruction after the RESET condition and before acknowledging the NMi. Processing then continues as usual.
A Z80-Based System Using the DMA With the SIO

Application Brief

January 1981

INTRODUCTION

In certain applications, serial data communications can be handled more efficiently by using a DMA device in conjunction with a serial controller. This application brief describes the use of the Z80A SIO and Z80A DMA hardware and software in a Z80-based system to transfer data to the SIO via the DMA.

Transfers through a serial data medium are usually done with a serial controller device, often a Universal Synchronous/Asynchronous Receiver/Transmitter (USART), such as the Z80 SIO. Additionally, some sort of controlling device is required to manipulate the data on a character-by-character basis, (usually a CPU). Transferring characters can be accomplished either by polling the USART, which forces the CPU to take time away from other activities, or by initiating an interrupt mechanism, which requires CPU time only if there is data to be moved. However, when large blocks of data need to be moved, even the interrupt mechanism becomes awkward. In these cases, a Direct Memory Access (DMA) device is especially valuable.

With DMA transfer, data is moved directly between memory and I/O (or additional memory) without CPU intervention. Once initiated by the CPU, DMA operation continues transparently to CPU operation until completed. Then the DMA device can either interrupt the CPU or restart its cycle using the previously programmed parameters.

HARDWARE DESCRIPTION

The hardware used in the example for this brief consists of a Z80A CPU, a Z80A DMA controller, a Z80A SIO/2, some RAM and ROM, and some support circuitry (Figure 1).

The Z80A DMA contains a 16-bit address bus, an 8-bit data bus, and 13 control lines for external interfacing. The Z80 DMA can generate independent addresses for Port A and Port B. Each address can be variable or fixed. Variable addresses can be programmed to either increment or decrement from the programmed starting addresses, whereas fixed addressing eliminates the need for separate enabling lines for I/O ports.

Readable registers contain the current address of each port and a count of the number of bytes searched and/or transferred. Additional registers allow the DMA to perform bit-maskable data comparisons on the data that is being searched and/or transferred. The DMA has 21 writeable control registers and seven readable status registers, which together provide a high degree of programmability.

The DMA function described is for a simple test operation using memory-to-I/O transfer with no search options. The DMA is initialized to transfer data from a pattern in memory to the SIO when the SIO requests a byte via the WAIT/READY signal line. The SIO then sends the byte to a terminal, which displays it for visual inspection. After a block of bytes has been sent, the DMA restarts itself (Auto Restart mode) and the process repeats continuously. Since the data pattern in memory consists of displayable ASCII characters, data is easily verified by observing the characters displayed on the terminal.

One feature of the Z80 DMA is the ease with which it interfaces with the Z80 CPU. The DMA is designed to connect directly to the CPU, as illustrated in Figure 2. The 16 address lines, eight data lines, and seven control lines are connected directly to the corresponding lines on the Z80 CPU. These signals are then buffered by the 74LS241s and distributed to the rest of the system. The data bus is buffered by the 74LS245 bidirectional octal buffer. Other connections to the DMA include clock, CE/WAIT, INT, RDY and IE1.

The clock input to the DMA is sensitive to both level and rise and fall times. The voltage should be no greater than +0.45V for a low level and no less than Vcc-0.6V for a
high level. Additionally, the rise and fall times for the waveform should be no greater than 30ns, according to the device specifications. A clock driver device is used to deliver the proper voltage levels and rise/fall times.

Figure 1. Block Diagram of a Z80 System with DMA and S10.

Figure 2. Schematic of CPU and DMA Interface
With the clock slightly more complex, the CE/WAIT input can be programmed as a WAIT control line for the DMA, similar to the WAIT input on the Z80 CPU. Figure 3 shows the gating that determines the CE/WAIT function.

The CE/WAIT input to the DMA serves a dual purpose. When the DMA is idle (Bus Acknowledge Input (BAI) inactive), the CE/WAIT input is used to select the DMA during a CPU access cycle, allowing the DMA to be treated as a peripheral device by the CPU. However, when the DMA takes control of the system bus, the CE/WAIT input is used to control the DMA, similar to the WAIT input on the Z80 CPU. Figure 3 shows the gating that determines the CE/WAIT function.

The WAIT/RDY pin on the SIO is connected to the RDY input on the DMA. This provides character transfer control between the SIO and DMA. In this application, the ready function is used and the WAIT/RDY pin is wired directly to the RDY input on the DMA with a pullup resistor. A low level initiates a DMA character transfer from memory to the SIO. The SIO drives the WAIT/RDY line High or Low so that pullup is not strictly required. However, upon reset, the SIO WAIT/RDY pin floats until the ready function is programmed in the SIO. Figure 4 shows the Z80 CPU-SIO interface.

Since the SIO has only one WAIT/RDY pin per channel, it can be used with the DMA only during transmit or receive but not both simultaneously. Therefore, characters received by the SIO are transferred via interrupts with the CPU intervening. The interrupt system also handles errors detected either during reception or when the SIO notices an external or status change.
Before any action can occur, initialization must be performed on the Z80 CPU, the DMA, and the SIO devices. Since interrupts are used in processing special SIO conditions, the Z80 CPU must be initialized for the proper interrupt mode. In the example, the CPU is set to Interrupt Mode 2 using the IN instruction. The upper eight bits of the interrupt vector are loaded into the I register via the A register in the CPU. The Stack Pointer (SP) register must be loaded by the program upon reset, because it has an undefined value. The SP register is used when processing interrupts and when the CALL instruction is executed during initialization. The appendix contains a source listing for a DMA test program using the SIO. The DMA is initialized for memory-to-I/O, byte-at-a-time transfer with the search option disabled and operates continuously until stopped by a command from the CPU. The program uses Port A of the DMA for the memory source address (SRC) and Port B for the destination address (DST) and utilizes the auto restart option on the DMA so that data can be sent to the terminal as a stream of characters. Since Port B is a fixed destination address, it must be declared as the source when the DMA is given the Load command (WR6, CFH), as stated in the programming section of the DMA Technical Manual (document number 00-2013-A). Table 1 shows the initialization sequence for the example described here.

The SIO initialization sequence is straightforward. The example uses channel A in Asynchronous Communication mode with the DMA providing data characters to the S10 on a transmit buffer empty condition. The terminal requires async format, two stop bits, and even parity. An external 1X clock is used with the S10 for the bit rate clock. The lower eight bits of the S10 interrupt vector are loaded into WR2 through channel B, and the Status Affects Vector (SAV) bit in WR1 is also set. SAV provides eight separate interrupt vectors (four for each channel), allowing easy program operation. Table 2 shows the programming sequence and mode of the S10 for DMA operation. Note that when DMA transfers are used to move data, the transmit buffer empty interrupt should not be enabled (WR1, bit 1=0).

A data test pattern is generated in the memory buffer area used for transmission to the S10 so that intelligible information can be sent to the terminal for easy verification. This is done by a short routine that fills the memory block with an incremental pattern of ASCII characters in the range of 20H to 7FH and appends a carriage return and a linefeed to the data block. Figure 5 contains a listing of the routine involved. The block length programmed into the DMA is one less than the actual block length transferred due to the counter characteristics of the Z80 DMA.

**Table 1. DMA Initialization Sequence**

1. Disable DMA
2. Issue six reset commands (insures a reset if DMA in undefined state)
3. WR0 - Port A (source) characteristics
4. Port A start address - low byte
5. Port A start address - high byte
6. Port A block length - low byte
7. Port A block length - high byte
8. WR1 - Port A increment address
9. WR2 - Port B is fixed address, I/O
10. WR4 - Byte mode, Port B address (low byte) follows
11. Port B (destination) address
12. WR5 - Auto Restart mode, CE/WAIT is multiplexed
13. Insure Port A is standard timing
14. Insure Port B is standard timing
15. Load Port B
16. WR0 - Port A is source, Port B is destination
17. Load Port A

**Table 2. SIO Initialization Sequence**

**Channel A**

1. Channel Reset
2. WR1 - WAIT/RDY enable for TX, ready function, RX interrupt on all characters; parity affects vector
3. WR4 - XI clock, two stop bits, even parity
4. WR5 - DTR, RTS active, TX seven bits, enable TX
5. WR3 - RX seven bits

**Channel B**

1. Channel Reset
2. WR1 - status affects vector
3. WR2 - lower eight bits of vector
Once the CPU, DMA, and SIO are set up, the program enables the DMA device (WR6, 87H) and the data transfer process begins. The SIO brings the WAIT/RDY output active as soon as the SIO has been initialized so that characters can be transmitted immediately. The user must insure that the DMA and data block have been set up properly before any data transfer actually occurs. DMA data transfer is different from the interrupt data transfer of the SIO, because with interrupts the SIO does not request data until it is activated by having a character sent to it.

Once operation of the DMA and SIO has begun, data transfers occur without CPU intervention unless the SIO encounters an error condition. An error causes the SIO to interrupt the CPU, thereby intervening in CPU processing. In this event, the CPU is interrupted by the device detecting the error and the DMA processing is terminated by the CPU. This termination is achieved by writing a command word to the DMA. The DMA remains disabled until given a command that enables it.

```assembly
LD HL, SRC ;%HL = start address
LD BC, BLKSIZE-2 ;%BC = length
LD D, 20H ;%D = data byte
LOOP:
LD (HL), D ;store character
INC D ;increment character code
AND 7FH ;mask upper bit
OR 20H ;keep displayable character
LD D, A ;save in %D
INC HL ;Bump memory ptr.
DEC BC ;Bump byte count
LD A, B ;see if through
JR NZ, LOOP ;no-loop
LD (HL), 13 ;CR
LD (HL), 10 ;LF
```

**Figure 5. Data Test Pattern Generator Routine Listing.**

CONCLUSION

This example shows only one aspect of using the DMA with the SIO. Use of the DMA with the SIO during receive deserves special consideration. Since the DMA operates without CPU processing, data received by the SIO does not normally indicate when the end of a message occurs. One solution to this problem is to send fixed-length data blocks so that the CPU can be interrupted when the DMA reaches terminal count. This is done by programming a fixed-length block count into the DMA and enabling it to interrupt the CPU upon End-Of-Block (EOB). As an alternative to the terminal count interrupt, the SIO can be programmed to interrupt the CPU when the closing flag is detected in SDLC mode. This allows the CPU to detect the end of a message using the SIO instead of the DMA.

Another method of detecting the end of a message is to dedicate a special EOB character used to terminate all message blocks. The DMA can then be programmed to search for this character during data transfers and to interrupt the CPU when the character is detected. This method allows for variable-length message blocks, up to the maximum byte count the DMA will accommodate. The disadvantage with this method is that the user must dedicate one character as the special EOB character.

The unique features of the DMA and SIO combine to form a powerful and flexible data communication mechanism. Due to the designed-in compatibility of the SIO and DMA, interfacing with both in hardware and software becomes a simplified task. Programming is easy because very little CPU intervention is necessary after initialization. Thus, the user is afforded a powerful tool for implementing an efficient, cost-effective data processing system.

APPENDIX

Following is a printout of the DMA/SIO test program. This program uses the DMA to transfer data from a pattern in memory to the SIO, which then sends the data, in async format at 9600 baud, to a terminal for display. The process continues until it is externally interrupted, such as by a reset. Interrupts are used to process error conditions or to receive characters. However, no code is shown that handles the characters once they are received. Error conditions are reset by the interrupt service routine, although nothing is shown for these conditions either. The user normally sets a condition flag after resetting the error condition, so that the driver program can determine the appropriate course of action.
DMA/SIO TEST PROGRAM

By M. PITCHER - 10/10/80

Generates block of data in RAM, then outputs to SIO via DMA, then continues forever.

RAM: EQU 2000H
RAMSIZE: EQU 1000H
SIO: EQU 0
SIOA: EQU SIOA+1
SIOB: EQU SIOB+1
DMA: EQU OFOH
DST: EQU SIOA
BLKSIZ: EQU 64
DMABLK: EQU BLKSIZ-1

START DMA AFTER INITIALIZATION (WR6, 37H)

DMA PARAMETERS

DMAWRO: EQU 0
XFER: EQU 1
SRCH: EQU 2
XFRSCH: EQU 3
A_B: EQU 4
ALSTA: EQU 8
AHSTA: EQU 10H
ALBLEN: EQU 20H
AHBLEN: EQU 40H

DMAWR1: EQU 4
AIO: EQU 8
AINCR: EQU 10H
ADECR: EQU 0
AFIXED: EQU 20H
AVTIM: EQU 40H

DMAWR2: EQU 0
BIO: EQU 8
BINCR: EQU 10H
BDECR: EQU 0
BFIXED: EQU 20H
BVTIM: EQU 40H

DMAWR3: EQU 80H
DMAEN: EQU 40H
INTEN: EQU 20H
MCHBYT: EQU 10H
MSKBYT: EQU 8
SOMCH: EQU 4

DMAWR4: EQU B1H
BYTE: EQU 0
CONT: EQU 20H
BURST: EQU 40H
ICB: EQU 10H
INTRDY: EQU 40H
DMASAV: EQU 20H
IV: EQU 10H
PCB: EQU 8
PULSE: EQU 4
INTEOB: EQU 2
INTMCH: EQU 1
BHSTA: EQU 8
BLSTA: EQU 4
PULSE: EQU 4
INTMCH: EQU 1

DMAWR5: EQU B2H
SETUP FOR ASYNC FORMAT AS FOLLOWS

- 9600 BAUD
- 2 STOP BITS
- 7 BIT CHARACTERS
- EVEN PARITY

PROGRAM ASSUMES DMA XFER OF TX DATA
THERE IS NO RECV DATA XFER
STATUS IS REFLECTED IN "SIOFLG" LOC
EXTERNAL TX AND RX CLOCK ASSUMED

SIOFLG - X X 1 1 X X 1 1
ERROR ASLEEP ERROR ASLEEP

CHANNEL B CHANNEL A

ASIOW: EQU 0
CHRES: EQU 18H
ESCRE: EQU 10H
TBERES: EQU 28H
SRCA: EQU 30H
TCR: EQU 80H
EDMRES: EQU 0COH

SIOWR1: EQU 1
WREN: EQU 80H
RDY: EQU 40H
WRDR: EQU 20H
RXIFC: EQU 8
RXIAP: EQU 10H
RXIA: EQU 18H
SIDOSAV: EQU 4
TXI: EQU 2
EXTI: EQU 1

SIOWR2: EQU 2

SIOWR3: EQU 3
RX8: EQU 0COH
RX6: EQU 80H
RX7: EQU 40H
RX5: EQU 0
AUTOEN: EQU 20H
HUNT: EQU 10H
RXRC: EQU 8

SIOWR4: EQU 4
X64: EQU 0COH
X32: EQU 80H
X16: EQU 40H
X1: EQU 0
EXTSYN: EQU 30H
SDLC: EQU 20H
SYN16: EQU 10H
SYNB: EQU 0
STOP2: EQU 0CH
STOP15: EQU 8
STOP1: EQU 4
SYNEN: EQU 0
EVEN: EQU 2
PARITY: EQU 1

SIOWR5: EQU 5
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ CODE</th>
<th>M</th>
<th>STMT</th>
<th>SOURCE</th>
<th>DMASIO</th>
<th>STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>143</td>
<td>DTR:</td>
<td>EQU</td>
<td>80H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>144</td>
<td>TX8:</td>
<td>EQU</td>
<td>60H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>145</td>
<td>TX6:</td>
<td>EQU</td>
<td>40H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>146</td>
<td>TX7:</td>
<td>EQU</td>
<td>20H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>147</td>
<td>TX5:</td>
<td>EQU</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>148</td>
<td>BREAK:</td>
<td>EQU</td>
<td>10H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>149</td>
<td>TXEN:</td>
<td>EQU</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>CRC16:</td>
<td>EQU</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>151</td>
<td>RTS:</td>
<td>EQU</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>152</td>
<td>TXCRC:</td>
<td>EQU</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>153</td>
<td>SICWR6:</td>
<td>EQU</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>154</td>
<td>SICWR7:</td>
<td>EQU</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>155</td>
<td>*EJ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>156</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*** MAIN PROGRAM ***</td>
</tr>
<tr>
<td>160</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**; i

**; MAIN PROGRAM**

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ CODE</th>
<th>M</th>
<th>STMT</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>ORG 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>JP BEGIN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>ORG * AND, OFFFOH OR 10H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0020</td>
<td>LD SP, STAK; INIT SP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0023</td>
<td>LD IM 2; INTERRUPT MODE 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0025</td>
<td>LD A, INVEC/256</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0027</td>
<td>LD I, A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0029</td>
<td>CALL INIT; INIT DMA, SIO</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>002C</td>
<td>LD HL, SRC; GENERATE DATA PATTERN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0032</td>
<td>LD D, 20H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0034</td>
<td>LD (HL), D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0035</td>
<td>INC D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0036</td>
<td>LD A, D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0037</td>
<td>AND 7FH</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0039</td>
<td>OR 20H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>003B</td>
<td>LD D, A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>003C</td>
<td>INC HL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>003D</td>
<td>DEC BC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>003E</td>
<td>LD A, B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>003F</td>
<td>OR C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0040</td>
<td>JR NZ, LOOP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0042</td>
<td>LD (HL), 13; CR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0044</td>
<td>INC HL</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0045</td>
<td>LD (HL), 10; LF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0047</td>
<td>LD A, 87H; ENABLE DMA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0049</td>
<td>OUT (DMA), A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>004B</td>
<td>JR $; LOOP FOREVER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0055</td>
<td>INIT DMA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0056</td>
<td>INIT DMA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0059</td>
<td>LD HL, SIOTA; INIT SID CH A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>005B</td>
<td>LD B, SIDEA-SIOTA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOC</td>
<td>OBJ CODE M</td>
<td>STMNT</td>
<td>SOURCE STATEMENT</td>
<td>ASM 5.9</td>
</tr>
<tr>
<td>------</td>
<td>------------</td>
<td>-------</td>
<td>------------------</td>
<td>--------</td>
</tr>
<tr>
<td>005D</td>
<td>EDB3</td>
<td>215</td>
<td>OTIR</td>
<td></td>
</tr>
<tr>
<td>005F</td>
<td>AF</td>
<td>216</td>
<td>XOR A</td>
<td>CLEAR SIOFLG</td>
</tr>
<tr>
<td>0060</td>
<td>320020</td>
<td>217</td>
<td>LD (SIOFLG),A</td>
<td></td>
</tr>
<tr>
<td>0063</td>
<td>C9</td>
<td>218</td>
<td>RET</td>
<td></td>
</tr>
</tbody>
</table>

219 *EJ

, INTERRUPT SERVICE ROUTINES

220

221

0064 CDD800 222 CALL SAVE ;CH. B TX BUFFER EMPTY

0067 3E00 225 LD A, SIOWRO

0069 D303 226 OUT (SIOCB),A

0068 3E28 227 LD A, TBERES

006D D303 228 OUT (SIOCB),A

006F C9 229 RET

230

231

0070 CDD800 232 CALL SAVE ;CH. B RX CHAR AVAIL

0073 DB02 233 IN A, (SIODB)

0075 C9 234 RET

235

236

0076 CDD800 237 CALL SAVE ,EXTERNAL/STATUS CHG

0079 3E00 238 LD A, SIOWRO

007B D303 239 OUT (SIOCB),A

007D 3E10 240 LD A, SRES

007F D303 241 OUT (SIOCB),A

0081 3A0020 242 LD A, (SIOFLG)

0084 CBE7 243 SET 4, A

0086 320020 244 LD (SIOFLG),A

0089 C9 245 RET

246

247

008A CDD800 248 CALL SAVE ;CH. B SPECIAL RX COND

008B 3E00 249 LD A, SIOWRO

008F D303 250 OUT (SIOCB),A

0091 3E30 251 LD A, SRCRES

0093 D303 252 OUT (SIOCB),A

0095 3A0020 253 LD A, (SIOFLG)

0098 CBEF 254 SET 5, A

009A 320020 255 LD (SIOFLG),A

009D C9 256 RET

257

258

009E CDD800 259 CALL SAVE ;CH. A TX BUFFER EMPTY

00A1 3E00 260 LD A, SIOWRO

00A3 D301 261 OUT (SIOCA),A

00A5 3E28 262 LD A, TBERES

00A7 D301 263 OUT (SIOCA),A

00A9 C9 264 RET

265

266

00AA CDD800 267 CALL SAVE ,CH. A RX CHAR AVAIL.

00AD DB00 268 IN A, (SIODA)

00AF C9 269 RET

270

271

00B0 CDD800 272 CALL SAVE ,EXTERNAL/STATUS CHG

00B3 3E00 273 LD A, SIOWRO

00B5 D301 274 OUT (SIOCA),A

00B7 3E10 275 LD A, SRCRES

00B9 D301 276 OUT (SIOCA),A

00BB 3A0020 277 LD A, (SIOFLG)

00BE CBE7 278 SET 0, A

00C0 320020 279 LD (SIOFLG),A

00C3 C9 280 RET

281

282

00C4 CDD800 283 CALL SAVE ,CH. A SPECIAL RX COND.

00C7 3E00 284 LD A, SIOWRO

00C9 D301 285 OUT (SIOCA),A
MATHEWS SAVE REGISTER ROUTINE

```
00D8  E3  296  EX  (SP),HL  ; SP = HL
00D9  D5  297  PUSH DE  ; DE
00DA  C5  298  PUSH BC  ; BC
00DB  F5  299  PUSH AF  ; AF
00DC  DDE5 300  PUSH IX  ; IX
00DE  FDE5 301  PUSH IY  ; IY
00E0  CDEEOO 302  CALL 00H  ; SAVE PC
00E3  FDE1 303  POP IY
00E5  DDE1 304  POP IX
00E7  F1  305  POP AF
00E8  C1  306  POP BC
00E9  D1  307  POP DE
00EA  E1  308  POP HL
00EB  FB  309  E1
00EC  ED4D 310  RETI
00EE  E9  313  JP (HL)
```

*EJ

; CONSTANTS

```
00EF  B3  319  DEFB  B3H  ; WR6, DISABLE DMA
00F0  C3  320  DEFB  OC3H  ; WR6, RESET
00F1  C3  321  DEFB  OC3H  ; WR6, RESET
00F2  C3  322  DEFB  OC3H  ; WR6, RESET
00F3  C3  323  DEFB  OC3H  ; WR6, RESET
00F4  C3  324  DEFB  OC3H  ; WR6, RESET
00F5  C3  325  DEFB  OC3H  ; WR6, RESET
00F6  79  326  DEFB  DMAWR0+XFER+ALSTA+AHSTA+ALBLEN+AHBLEN
00F7  01  327  DEFB  SRC.AND.255  ; PORT A ADDR (L)
00F8  20  328  DEFB  SRC/256  ; PORT A ADDR (H)
00F9  3F  329  DEFB  DMABLK.AND.255  ; PORT A COUNT (L)
00FA  00  330  DEFB  DMABLK/256  ; PORT A COUNT (H)
00FB  14  331  DEFB  DMAWR1+AINC
00FC  2B  332  DEFB  DMAWR2+BIO+BFIXED
00FD  B5  333  DEFB  DMAWR4+BYTE+BLSTA
00FE  00  334  DEFB  DST.AND.255  ; PORT B ADDR (L)
00FF  B2  335  DEFB  DMAWR5+AUTORS+CEWAIT
0100  C7  336  DEFB  OC7H  ; WR6, RESET A TIMING
0101  CB  337  DEFB  OCBH  ; WR6, RESET B TIMING
0102  CF  338  DEFB  OCFH  ; WR6, LOAD PORT B
0103  05  339  DEFB  DMAWR0+XFER+A_B  ; A -> B
0104  CF  340  DEFB  OCFH  ; WR6, LOAD COUNTERS
```

DMAEND: EQU $  

```
0105  00  344  DEFB  SIDWRO  ; CH. RESET
0106  1B  345  DEFB  CHRES
0107  01  346  DEFB  SIDWR1  ; RDY/WAIT, INT. MODE
0108  D0  347  DEFB  WREN+RDY+RXIAP
0109  04  348  DEFB  SIDWR4  ; MODE
010A  0F  349  DEFB  X1+STOP2+EVEN+PARITY
010B  05  350  DEFB  SIDWR5  ; TX PARAMS.
010C  AA  351  DEFB  DTR+TX7+TXEN+RTS
010D  03  352  DEFB  SIDWR3  ; RX PARAMS.
010E  40  353  DEFB  RX7
```

SIDA: EQU $  

```
010F  354  DEFB  SIDWRA  ; RXIN
0110  355  DEFB  SIDWRB  ; RXOUT
```

751-1809-0002  2-44  2-6-81
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ</th>
<th>CODE</th>
<th>M</th>
<th>STMT</th>
<th>SOURCE</th>
<th>STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>010F</td>
<td>00</td>
<td>357</td>
<td></td>
<td></td>
<td></td>
<td>DEF8 S1OWRO</td>
</tr>
<tr>
<td>0110</td>
<td>18</td>
<td>358</td>
<td></td>
<td></td>
<td></td>
<td>DEF8 CHRES</td>
</tr>
<tr>
<td>0111</td>
<td>01</td>
<td>359</td>
<td></td>
<td></td>
<td></td>
<td>DEF8 S1OWR1</td>
</tr>
<tr>
<td>0112</td>
<td>04</td>
<td>360</td>
<td></td>
<td></td>
<td></td>
<td>DEF8 S1OSAV</td>
</tr>
<tr>
<td>0113</td>
<td>02</td>
<td>361</td>
<td></td>
<td></td>
<td></td>
<td>DEF8 S1OWR2</td>
</tr>
<tr>
<td>0114</td>
<td>10</td>
<td>362</td>
<td></td>
<td></td>
<td></td>
<td>DEF8 S1ОVEC AND.255</td>
</tr>
<tr>
<td></td>
<td>363</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>SID8E. EQU $</td>
</tr>
<tr>
<td></td>
<td>364</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*EJ</td>
</tr>
<tr>
<td></td>
<td>365</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>366</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>: DATA AREA</td>
</tr>
<tr>
<td></td>
<td>367</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>368</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>ORG RAM</td>
</tr>
<tr>
<td>2001</td>
<td>369</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S1ОFLG: DEFS 1</td>
</tr>
<tr>
<td>2041</td>
<td>370</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S1ОMA SOURCE ADDR</td>
</tr>
<tr>
<td></td>
<td>371</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DEFS 64</td>
</tr>
<tr>
<td></td>
<td>372</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>STAK: EQU $</td>
</tr>
<tr>
<td></td>
<td>373</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>374</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>END</td>
</tr>
</tbody>
</table>
Introduction.

The Z80 Serial Input/Output (SIO) controller is designed for use in a wide variety of serial-to-parallel input and parallel-to-serial output applications. In this application note, only asynchronous applications are considered. The emphasis is almost completely on software implementation, with only modest reference to hardware considerations.

While reference is made only to the Z80 SIO, the entire text also applies to the Z80 DART, which is functionally identical to the Z80 SIO in asynchronous applications.

Communication, either on an external data link or to a local peripheral, occurs in one of two basic formats: synchronous or asynchronous. In synchronous communication, a message is sent as a continuous string of characters where the string is preceded and terminated by control characters; the preceding control characters are used by the receiving device to synchronize its clock with the transmitter's clock. In asynchronous communication, which is described in this application note, there is no attempt at synchronizing the clocks on the transmitting and receiving devices. Instead, each fixed-length character (rather than character string) is preceded and terminated by "framing bits" that identify the beginning and end of the character. The time between bits within a character is approximately constant, since the clocks or "baud rates" in the transmitter and receiver are selected to be the same, but the time between characters can vary.

Thus, in asynchronous communication, each character to be transmitted is preceded by a "start" framing bit and followed by one or more "stop" framing bits. A start bit is a logical 0 and a stop bit is a logical 1. The receiver will look for a start bit, assemble the character up to the number of bits the SIO has been programmed for, and then expect to find a stop bit. The time between the start and stop bits is approximately constant, but the time between characters can vary. When one character ends, the receiving device will wait idly for the start of the next character while the transmitter continues to send stop or "marking" bits (both the stop bits and the marking bits are logical 1). Figure 1 illustrates this. A very common application of asynchronous communication is with keyboard devices, where the time between the operator's keystrokes can vary considerably.

---

**Figure 1. Asynchronous Data Format**
If the transmitter's clock is slightly faster than the receiver's clock, the transmitter can be programmed to send additional stop bits, which will allow the receiver to catch up. If the receiver runs slightly faster than the transmitter, then the receiver will see somewhat larger gaps between characters than the transmitter does, but the characters will normally still be received properly. This tolerance of minor frequency deviations is an important advantage of using asynchronous I/O. Note however that errors, called "framing errors," can still occur if the transmitter and receiver differ substantially in speed, since data bits may then be erroneously treated as start or stop bits.

The SIO may be used in one of three modes: Polled, Interrupt, or Block Transfer, depending on the capabilities of the CPU. In Polled mode the CPU reads a status register in the SIO periodically to determine if a data character has been received or is ready for transmission. When the SIO is ready, the CPU handles the transfer within its main program.

In Interrupt mode, which is far more common, the SIO informs the CPU via an interrupt signal that a single-character transfer is required. To accomplish this, the CPU must be able to check for the presence of interrupt signals (or "interrupt requests") at the end of most instruction cycles. When the CPU detects an interrupt it branches to an interrupt service routine which handles the single-character transfer. The beginning memory address of this interrupt service routine can be derived, in part, from an "interrupt vector" (8-bit byte) supplied by the SIO during the interrupt acknowledge cycle.

In Block Transfer mode, the SIO is used in conjunction with a DMA (direct memory access) controller or with the Z80 or Z8000 CPU block transfer instructions for very fast transfers. The SIO interrupts the CPU or DMA only when the first character of a message becomes available, and thereafter the SIO uses only its Wait/Ready output pin to signal its readiness for subsequent character transfers. Due to the faster transfer speeds achievable, Block Transfer mode is most commonly used in synchronous communication and only rarely in asynchronous formats. It is therefore not treated with specific examples in this application note.

Since Polled mode requires CPU overhead regardless of whether or not an I/O device desires attention, Interrupt mode is usually the preferred alternative when it is supported by the CPU. Note that the choice of Polled or Interrupt mode is independent of the choice of synchronous or asynchronous I/O. This latter choice is usually determined by the type of device to which the system is communicating.

The SIO comes in four different 40-pin configurations: SIO/0, SIO/1, SIO/2, and SIO/9. The first three of these support two independent full-duplex channels, each with separate control and status registers used by the CPU to write control bytes and read status bytes. The SIO/9 differs from the first three versions in that it supports only one full-duplex channel. The product specifications for these versions explain this in full.

There are 41 different signals needed for complete two-channel implementation in the SIO/0, SIO/1, and SIO/2, but only 40 pins are available. Therefore, the versions differ by either omitting one signal or bonding two signals together. The dual-channel asynchronous-only Z80 DART has the same pin configuration as the SIO/0.

The serial-to-parallel and parallel-to-serial conversions required for serial I/O are performed automatically by the SIO. The device is connected to a CPU by an 8-bit bidirectional data path, plus interrupt and I/O control signals.

The SIO was designed to interface easily to a Z80 CPU, as shown in Figure 2. Other microprocessors require a small amount of external logic to generate the necessary interface signals.

The SIO provides a sophisticated vectored-interrupt facility to signal events that require CPU intervention. The interrupt structure is based on the Z80 peripheral daisy chain. Non-Z80 microprocessors that are unable to utilize external vectored interrupts require some additional external logic to utilize efficiently this interrupt facility. Some non-Z80 system designs do not utilize the vectored interrupt structure of the SIO at all. Instead, these require the CPU to poll the SIO's status through the data bus or to use non-vectored SIO interrupts.

Microprocessors such as the 6808 and 6800 need some signal translation logic to generate SIO read/write and clock timing. CPU signals which synchronize a peripheral device read or write operation are gated to form the proper I/O signals for the SIO. The SIO is selected by some processor-dependent function of the address bus in a memory or I/O addressing space.
In the next section we begin with a discussion of features common to all forms of asynchronous I/O. This is followed by discussions of polled asynchronous I/O and interrupt asynchronous I/O. Next is a series of frequently asked questions about the SIO when used in asynchronous applications. Finally, an example of a simple interrupt-driven asynchronous application is given and discussed in detail. For a complete understanding of the material covered, the following publications are needed:

- Z80 SIO Product Specification or Z80 DART Product Specification
- Z80 SIO Technical Manual
- Z80 Family Program Interrupt Structure
- Z80 CPU Technical Manual
- Z80 Assembly Language Programming Manual

Figure 2. SIO Hardware Interfacing
SECTION 2

Character CPU-SIO

Addressing the SIO

The CPU must have a means to identify any specific I/O device, including any attached SIO. In a Z80 CPU environment, this is done by using the lower 8 bits of the address bus (A0-A7). Typically, the A1 bit is wired to the SIO's B/A input pin for selecting access to Channel A or Channel B, and the A0 bit is wired to the SIO's C/D input pin for selecting the use of the data bus as an avenue for transferring control/status information (C) or actual data messages (D). The remaining bits of the address bus, A2-A7, contain a port address that uniquely identifies the SIO.

Asynchronous Format Operations

Bits per Character. The SIO can receive or transmit 5, 6, 7, or 8 bits per character. This can be different for transmission and reception, and different for each channel. ASCII characters, for example, are usually transmitted as 7 bits. The SIO can in fact transmit fewer than 5 bits per character when set to the 5-bit mode; this is discussed further in the section entitled "Questions and Answers."

Parity. A parity bit is an additional bit added to a character for error checking. The parity bit is set to 0 or 1 in order to make the total number of 1s in the character (including parity bit) even or odd, depending on whether even or odd parity is selected. The SIO can be set either to add an optional parity bit to the "bits per character" described above, or not to add such a bit. When a parity bit is included, either even or odd parity can be chosen. This selection can be made independently for each channel.

Start and Stop Bits. There are two types of framing bits for each character: start and stop. When transmitting asynchronously, the SIO automatically inserts one start bit (logic 0) at the beginning of each character transmitted. The SIO can be programmed to set the number of stop bits inserted at the end of each character to either 1, 1½, or 2. The receiver always checks for 1 stop bit. Stop bits refer to the length of time that the stop value, a logic 1, will be transmitted; thus 1½ stop bits means that a 1 will be transmitted for the length of clock time that 1½ bits would normally take up. A logic 1 level that continues after the specified number of stop bits is called a "marking" condition or "mark bits."

CPU-SIO Character Transfers

The SIO always passes 8-bit bytes to the CPU for each character received, no matter how many “bits per character” are specified in the SIO initialization phase. If the number of “bits per character” is less than eight, parity and/or stop bits will be included in the byte sent to the CPU. The received character starts with the least-significant bit (D0) and continues to the most-significant bit; it is immediately followed by the parity bit (if parity is enabled) and by the stop bit, which will be logic 1 unless there is a framing error. The remainder of the byte, if space is still available, is filled with logic 1s (marking). If the “bits per character” is eight, then the byte sent to the CPU will contain only the data bits. In all cases, the start bit is stripped off by the SIO and is not transmitted to the CPU.

Clock Divider

The SIO has five input pins for clock signals. One of these inputs (CLK) is used only for internal timing and does not affect transmission or reception rates. The other four clock inputs (RxCA, TxCA, RxCB, and TxCB) are used for timing the reception and transmission rates in Channels A and B. Only these last four are involved in “clock dividing.” A clock divider within the SIO can be programmed to cause reception/transmission clocking at the actual input clock rate or at 1/16, 1/32, or 1/64 of the input clock rate. The receiver and transmitter clock divisions within a given channel must be the same, although their input clock rates can be different. The x1 clock rate can be used only if the transitions of the Receive clock are synchronized to occur during valid data bit times.
### Auto Enables

The SIO has an Auto Enables feature that allows automatic SIO response and telephone answering. When Auto Enables is set for a particular channel, a transition to logical 0 (Low input level) on the respective Data Carrier Detect (DCD) input will enable reception, and a transition to logical 0 on the respective Clear To Send (CTS) input will enable transmission. This is described below under the heading “Modem Control.”

### Special Receive Conditions

There are three error conditions that can occur when the SIO is receiving data. Each of these will cause a status bit to be set, and if operating in Interrupt mode, the SIO can optionally be programmed to interrupt the CPU on such an error. The error conditions are called “special receive conditions” and they include:

- **Framing error.** If a stop bit is not detected in its correct location after the parity bit (if used) or after the most-significant data bit (if parity is not used), a framing error will result. The start bit preceding the character’s data bits is not considered in determining a framing error, although character assembly will not begin until a start bit is detected.

- **Parity error.** If parity bits are attached by the external I/O device and checked by the SIO while receiving characters, a parity error will occur whenever the number of logic 1 data bits in the character (including the parity bit) does not correspond to the odd/even setting of the parity-checking function.

- **Receiver overrun error.** SIO buffers can hold up to three characters. If a character is received when the buffers are full (i.e., characters have not been read by the CPU), an SIO receiver overrun error will result. In this case, the most recently received character overwrites the next most recently received character.

### Modem Control

Five signal lines on the SIO are provided for optional modem control, although these lines can also be used for other general-purpose control functions. They are:

- **RTS (Request To Send).** An output from the SIO to tell its modem that the SIO is ready to transmit data.

- **DTR (Data Terminal Ready).** An output from the SIO to tell its modem that the SIO is ready to receive data.

- **CTS (Clear To Send).** An input to the SIO from its modem that enables SIO transmission if the Auto Enables function is used.

- **DCD (Data Carrier Detect).** An input to the SIO from its modem that enables SIO reception if the Auto Enables function is used.

### SYNC (Synchronization)

A spare input to the SIO in asynchronous applications. This input may be used for the Ring Indicator function, if necessary, or for general-purpose inputs.

In most applications of asynchronous I/O that use modems, the RTS and DTR control lines and the Auto Enables function are activated during the initialization sequence, and they are left active until no further I/O is expected. This causes the SIO to tell its modem continuously that the SIO is ready to transmit and receive data, and it allows the modem to enable automatically the SIO’s transmission and reception of data. Figure 3 illustrates this.

---

**Figure 3. Modem Control (Single Channel)**
External/Status
Interrupts
A change in the status of certain external inputs to the SIO will cause status bits in the SIO to be set. In the Polled Mode, these status bits can be read by the CPU. In the Interrupt mode, the SIO can also be programmed to interrupt the CPU when the change occurs. There are three such “external/status” conditions that can cause these events:

- **DCD.** Reflects the value of the DCD input.
- **CTS.** Reflects the value of the CTS input.
- **Break.** A series of logic 0 or “spacing” bits.

Initialization
The SIO contains eight write registers for Channel B (WR0-WR7) and seven write registers for Channel A (all except write register WR2). These are described fully in the Z80 SIO Technical Manual and are summarized in Appendix B. The registers are programmed separately for each channel to configure the functional personality of the channel. WR2 exists only in the Channel B register set and contains the interrupt vector for both channels. Bits in each register are named D7 (most significant) through D0. With the exception of WR0, programming the write registers requires two bytes: the first byte is to WR0 and contains pointer bits for selection of one of the other registers; the second byte is written to the register selected. WR0 is a special case in that all of the basic commands can be written to it with a single byte.

There are also three read registers, named RR0 through RR2, from which status results of operations can be read by the CPU (see Appendix B). Both channels have a set of read registers, but register RR2 exists only in Channel B.

Let us now look at the typical sequence of write registers that are loaded to initialize the SIO for either Polled or Interrupt-driven asynchronous I/O. Figure 4 illustrates the sequence. Except for step E, this loading is done for each channel when both are used. Steps E and F are described further in the section on “Interrupt-Driven Environments.”

Registers WR6 and WR7 are not used in asynchronous I/O. They apply only to synchronous communication.

The related publications on the SIO should be referred to at this point. They will be necessary in following the discussion of functions. In particular, the following material should be reviewed:


---

**Figure 4. Typical Initialization Sequence (One Channel)**

Steps A through F are performed in sequence

*Channel B only

Interrupt mode only Polling mode begins I/O after step D

**NOTES**
Polled Environments.

In a typical Polled environment, the SIO is initialized and then periodically checked for completion of an I/O operation. Of course, if the checking is not frequent enough, received characters may be lost or the transmitter may be operated at a slower data rate than that of which it is capable. Initialization for Polled I/O follows the general outline described in the last section. We now give an overview of routines necessary for the CPU to check whether a character has been received by the SIO or whether the SIO is ready to transmit a character.

To check whether a character has been received, and to obtain a received character if one is available, the sequence illustrated in Figure 5 should be followed after the SIO is initialized. We assume that reception was enabled during initialization; if it was not, the Rx Enable bit in register WR3 must be turned on before reception can occur. This must be done for each channel to be checked.

Bit D0 of register RR0 is set to 1 by the SIO if there is at least one character available to be received. The SIO contains a three-character input buffer for each channel, so more than one character may be available to be received. Removing the last available character from the read buffer for a particular channel turns off bit D0.

If bit D0 of register RR0 is 0, then no character is available to be received. In this case it is recommended that checks be made of bit D7 to determine if a Break sequence (null character plus a framing error) has been received. If so, a Reset External/Status Interrupts command should be given; this will set the External/Status bits in register RR0 to the values of the signals currently being received. Thus, if the Break sequence has terminated, the next check of bit D7 will so indicate. It may also be desirable to check bit 3 of register RR0 which reports the value of the Data Carrier Detect (DCD) bit.

In any case, if bit D0 of register RR0 is 0, polled receive processing terminates with no character to receive. Depending on the facilities of the associated CPU, this step may be repeated until a character is available (or possibly a time-out occurs), or the CPU may return to other tasks and repeat this process later.

If bit D0 of register RR0 is 1, then at least one character is available to be read. In this case, the value of register RR1 should first be read and stored to avoid losing any error information (the manner in which it is read is explained later). The character in the data register is then read. Note that the character must be read to clear the buffer even if there is an error found.

Finally, it is necessary to check the value stored from register RR1 to determine if the character received was valid. Up to three bits need to be checked: bit 6 is set to 1 for a framing error, bit 5 is set to 1 for a receiver overrun error (which occurs when the receive buffers are overwritten, i.e., no character has been removed and more than three characters have been received), and bit 4 is set to 1 for a parity error (if parity is enabled at initialization time). In case of a receiver overrun or parity error, an Error Reset command must be given to reset the bits.
Character Transmission

To check that an initialized SIO is ready to transmit a character on a channel, and if so to transmit the character, the steps illustrated in Figure 6 should be followed. We assume that the Request To Send (RTS) bit in WR5, if required by the external receiving device, and the Transmit (Tx) Enable bit were set at initialization.

Depending on the external receiving device, the following bits in register RRO should be checked: bit 3 (DCD), to determine if a data carrier has been detected; bit 5 (CTS), to determine if the device has signalled that it is clear to send; and bit 7 (Break), to determine if a Break sequence has been received. If any of these situations have occurred, the bits in register RRO must be reset by sending the Reset External/Status Interrupts command, and the transmit sequence must be started again.

Next, bit 2 of register RRO is checked. If this bit is 0, then the transmit buffer is not empty and a new character cannot yet be transmitted. Depending on the capabilities of the CPU, this is repeated until a character can be transmitted (or a timeout occurs), or the CPU may return to other tasks and start again later.

If bit 2 of register RRO is 1, then the transmit buffer is empty and the CPU may pass the character to be transmitted to the SIO, completing the transmit processing. On the Z80 CPU, this is done with an OUT instruction to the SIO data port.

Assumptions for an Example

Now let us consider some examples in more detail. We assume we are given an external device to which we will input and output 8-bit characters, with odd parity, using the Auto Enables feature. We will support this device with I/O polling routines following the patterns illustrated in Figures 5 and 6. We assume that the CPU will provide space to receive characters from the SIO as fast as the characters are received by the SIO, and that the CPU will transfer characters as fast as the output can be accomplished by the SIO.

Initialization

We begin with the initialization code for the SIO. This follows the outline illustrated in Figure 4. In the following sample code, each time register WR0 is changed to point to another register, the Reset External/Status Interrupts command is given simultaneously. Whenever a transition on any of the external lines occurs, the bits reporting such a transition are latched until the Reset External/Status Interrupts command is given. Up to two transitions can be remembered by the SIO. Therefore, it is desirable to do at least two different Reset External/Status Interrupts commands as late as possible in the initialization so that the status bits reflect the most recent information. Since it doesn’t hurt, we include these commands each time WR0 is changed to point to another register. This is an easy way to code the initialization to insure that the appropriate resets occur.

In the example below, the logic states on the C/D control line and the system data bus (D7-D0) are illustrated, together with comments.

![Figure 6. Polled Transmit](image-url)
Initialization
(Continued)

<table>
<thead>
<tr>
<th>Bits sent to the SIO</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/D</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>1 0 0 0 0 1 1 0 0 0</td>
</tr>
<tr>
<td>1 0 0 0 0 1 0 1 0 0</td>
</tr>
</tbody>
</table>

Effects and Comments

Channel Reset command sent to register WR0 (D5-D3).

Point WR0 to WR4 (D2-D0) and issue a Reset External/Status Interrupts command (D5-D3). Throughout the initialization, whenever we point WR0 to another register, we will also issue this command for the reasons noted above.

Set WR4 to indicate the following parameters (from left to right):
A. Run at 1/64 the input clock rate (D7-D6).
B. Disable the sync bits and send out 2 stop bits per character (D5-D2).
C. Enable odd parity (D1-D0).

Point WR0 to WR3.

Set WR3 to indicate the following:
A. 8-bit characters to be received (D7-D6).
B. Auto Enables on (D5).
C. Receive (Rx) Enable on (D0).

Point WR0 to WR5.

Set WR5 to indicate the following:
A. Data Terminal Ready (DTR) on (D7).
B. 8-bit characters to be transmitted (D6-D5).
C. Break not to be transmitted (D4).
D. Transmit (Tx) Enable on (D3).
E. Request To Send (RTS) on (D1).

Reset and Error Sequences

In the receive and transmit routines that follow, we treat errors such as a transition on the Data Carrier Detect line by calling for a "reset sequence" to set the values in read register RRO to reflect the current values found at the pins. This sequence consists of giving the Reset External/Status Interrupts command and beginning the driver over again. The command takes the form of a write to register WR0:

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Permits the status bits in RRO to reflect current status.

This command does not turn off the latches for such things as parity errors stored in bits 4-6 of register RR1. When such an error occurs and the latches must be reset, we will call for an "error sequence." This sequence consists of giving the Error Reset command and beginning the driver over again. The command also takes the form of a write to register WR0:

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 1 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Resets the latches in register RR1.

When specifying the result of reading register RRO or RR1 or specifying data, we will indicate the values read as follows:

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>D  D  D  D  D  D  D  D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Read a byte from the designated register.

Receive and Transmit Routines

Now we will first give an example of the receive routine. This parallels the preceding discussion of "Character Reception."

The framing error in this routine is reported on a character-by-character basis and it is not necessary to execute an "error sequence" if it is the only error received. However, it is not harmful to do so.

Next, we give an example of transmission code that parallels the above discussion on "Character Transmission."
Effects and Comments (Receive Routine)

Read a byte from RR0 (the default read register); if \( D_0 = 0 \) then no character is ready to be received. In this case, if \( D_7 \) (Break) or \( D_3 \) (Data Carrier Detect) have changed state, then execute a "reset sequence." If \( D_0 = 0 \) and \( D_7 \) and \( D_3 \) have not changed state, then no character is ready to be received, either loop on this read or try again later.

Point WR0 to read from RR1, we will now check for errors in the character read. Note that Reset External/Status Interrupt Commands are not done normally to avoid losing a line-status change.

Read a byte from RR1; if either bit \( D_6 = 1 \) (framing error), \( D_6 = \) (receive overrun error), or \( D_4 = 1 \) (parity error), the character is invalid and an "error sequence" should be executed after the following step.

Read in the data byte received. This must be done to clear the SIO buffer even if an error is detected.

Effects and Comments (Transmit Routine)

Read a byte from RR0; if either bit \( D_3 \) (Data Carrier Detect), \( D_2 \) (Clear To Send) or \( D_7 \) (Break) have changed state, a "reset sequence" should be executed. If \( D_3 \), \( D_2 \) and \( D_7 \) have not changed state, then if \( D_2 = 0 \), the transmit buffer is not yet empty and a transmit cannot take place; either loop, reading RR0, or try again later.

Send the data byte to be transmitted.

### SECTION 4

**Interrupt-Driven Environments.**

In a typical interrupt-driven environment, the SIO is initialized and the first transmission, if any, is begun. Thereafter, further I/O is interrupt driven. When action by the CPU is needed, an SIO interrupt causes the CPU to branch to an interrupt service routine after the CPU first saves state information.

In common usage, if I/O is interrupt driven, all interrupts are enabled and each different type of interrupt is used to cause a CPU branch to a different memory address. There is perhaps one frequent exception to this: parity errors are sometimes checked only at the end of a sequence of characters. The SIO facilitates this kind of operation since the parity error bit in read register RR1 is latched; once the bit is set it is not reset until an explicit reset operation is done. Thus, if a parity error has occurred on any character since last reset, bit 4 in register RR1 will be set. It is then possible to set register WR1 so that parity errors do not cause an error interrupt when a character is received. The user then has the obligation to poll for the value of the parity bit upon completion of the sequence.

SIO initialization for Interrupt mode normally requires two steps not used in Polled mode: an interrupt vector (if used) must be stored in write register WR2 of Channel B and write register WR1 must be initialized to specify the form of interrupt handling. It is preferable to initialize the interrupt vector in WR2 first. In this way an interrupt that arrives after the enabling bits are set in WR1 will cause proper interrupt servicing.

### Interrupt Vectors

The interrupt vector, register WR2 of Channel B, is an 8-bit memory address. When an interrupt occurs (and note that an interrupt can only occur after interrupts have been enabled by writing to register WR1) the interrupt vector is normally taken as one byte of an address used by the CPU to find the location of the interrupt service routine. It is also possible to cause the particular type of interrupt condition to modify the address vector in WR2 before branching, resulting in a branch to a different memory location for each interrupt condition. This is a very useful construct; it permits short, special-purpose interrupt routines. The alternative, to have one general-purpose interrupt routine which must determine the situation before proceeding, can be quite inefficient. This is usually undesirable since the speed of interrupt-service routines is often a critical factor in determining system performance.
Interrupt Vectors (Continued)

There are at most eight different types of interrupts that the SIO may cause, four for each of the two channels. If bit 1 in register WR1 of Channel B has been turned on so that an interrupt will modify the interrupt vector, the three bits (1-3) of the vector will be changed to reflect the particular type of interrupt. These interrupts follow a hardware-set priority as follows, starting with the highest priority:

Channel A Special Receive Condition sets bits 3-1 of WR1 to 111,
Channel A Character Received sets bits 3-1 to 110,
Channel A Transmit Buffer Empty sets bits 3-1 to 100,
Channel A External/Status Transition sets bits 3-1 to 101.
Channel B Special Receive Condition sets bits 3-1 to 011,
Channel B Character Received sets bits 3-1 to 010,
Channel B Transmit Buffer Empty sets bits 3-1 to 000,
Channel B External/Status Transition sets bits 3-1 to 001.

For example, suppose that the interrupt vector had the value 11110001 and the Status Affects Vector bit is enabled, along with all interrupt-enable bits. When an External/Status transition occurs in Channel A, the three zeros (bits 3-1) would be modified to 101, yielding an interrupt vector of 11111011. The value of the interrupt vector, as modified, may be tained by reading register RR2 in Channel B.

Initialization

In general, the initialization procedure illustrated in Figure 4 can still be followed. All six steps (A through F) are required here. After completing the first four steps, which are the same as initialization for polled I/O, it is necessary to load an interrupt vector into WR2 of Channel B. Information is then written into register WR1 specifying which interrupts are to be enabled and whether a specific kind of interrupt should modify the interrupt vector.

Now let us give an example. As in the polled example, we assume that we are given a device to which we will input and output 8-bit characters, with odd parity, using the Auto Enables feature. We also assume the CPU will provide space to store characters as received.

We do not discuss the SIO commands and registers in detail. This is done in the Z80 SIO Technical Manual. A summary of the register bit assignments taken from the Z80 SIO Serial Input/Output Product Specification is included at the end of this note. Recall that to write a register other than register WR0, the number of the register to be written is first sent to register WR0, and the following byte will be sent to the named register. Similarly, to read a register other than RR0 (the default), the number of the register to be read is first written to register WR0 and the next byte read will return the contents of the register named.

In our example below, each time register WR0 is changed to point to another register, the Reset External/Status Interrupts command is also given. Whenever a transition on any of the external/status lines occurs, the bits reporting the transition are latched until the Reset External/Status Interrupts command is given. Up to two transitions can be remembered by the internal logic of the SIO. Therefore, it is desirable to do at least two different Reset External/Status Interrupt commands as possible in the initialization so that the status bits reflect the most recent information. Since it doesn’t hurt, we give these commands each
Initialization (Continued)

Time WR0 is changed to point to another register. This is an easy way to code the initialization to assure that the appropriate resets occur.

The columns below show the logic states on the C/D control line and the system data bus (D7-D0), together with comments.

<table>
<thead>
<tr>
<th>C/D</th>
<th>D7</th>
<th>D8</th>
<th>D9</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Effects and Comments

- **Channel Reset command sent to register WR0 (D3-D2).**
- **Point WR0 to WR4 (D2-D0) and issue a Reset External/Status Interrupts command (D3-D2).** Throughout the initialization, whenever we point WR0 to another register we will also issue a Reset External/Status Interrupts command for the reasons noted above.

Set WR4 to indicate the following parameters (from left to right):
- A. Run at 1/64 the clock rate (D7-D6).
- B. Disable the sync bits and send out 2 stop bits per character (D6-D2).
- C. Enable odd parity (D1-D0).

Point WR0 to WR3.

Set WR3 to indicate the following:
- A. 8-bit characters to be received (D7-D6).
- B. Auto Enables on (D3).
- C. Rx Enable on (D0).

Point WR0 to WR5.

Set WR5 to indicate the following:
- A. Data Terminal Ready (DTR) on (D7).
- B. 8-bit characters to be transmitted (D6-D2).
- C. Break not to be transmitted (D4).
- D. Tx Enable on (D2).
- E. Request To Send (RTS) on (D1).

Point WR0 to WR2 (Channel B only).

Set the interrupt vector to point to address 11100000 (which is hex 80 and decimal 224). Once interrupts are enabled, they will cause a branch to this memory location, modified as described above if the Status Affects Vector bit is turned on (which it will be here). This vector is only set for Channel B, but it applies to both channels. It has no effect when set in Channel A.

Point WR0 to WR1.

Set WR1 to indicate the following:
- A. Cause interrupts on all characters received, treating a parity error as a Special Receive Condition interrupt (D4-D3).
- B. Turn on the Status Affects Vector feature, causing interrupts to modify the status vector—meaningful only on Channel B, but will not hurt if set for Channel A (D2).
- C. Enable interrupts due to transmit buffer being empty (D1).
- D. Enable External/Status interrupts (D0).
A Special Receive Condition interrupt occurs (a) if a parity error has occurred, (b) if there is a receiver overrun error (data is being overwritten because the channel’s three-byte receiver buffer is full and a new character is being received), or (c) if there is a framing error. The processing in this case is the following:

1. Issue an Error Reset command (to register WRO) to reset the latches in register RR1.
2. Read the character from the read buffer and discard it to empty the buffer.

It may be desirable to read and store the value of register RR1 to gather statistics on performance or determine whether to accept the character. In some applications, a character may still be acceptable if received with a framing error.

In specifying the result of reading register RR0, RR1, or specifying data, we will indicate the values as follows:

<table>
<thead>
<tr>
<th>Bits sent and received</th>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/D</td>
<td>1 0 0 0 0 0 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 D D D D D D D D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 0 1 1 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0 D D D D D D D D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Effects and Comments

If we need to know what kind of error occurred, we point WRO to read from RR1. Note that the Reset External/Status Interrupts command is not used. This avoids losing a valid interrupt.

Read a byte from RR1; one or more of bit D6 (framing error), D5 (receive overrun error), or D4 (parity error) will be 1 to indicate the specific error.

Give an Error Reset command to reset all the error latches.

Read the data byte received. This must be done to clear the receiver buffer, but the character will generally be disregarded.

When an Rx Character Available interrupt occurs, the character need only be read from the read buffer and stored. If parity is enabled with character lengths of 5, 6, or 7 bits, the received parity bit will be transferred with the character. Any unused bits will be 1s.

To respond to an External/Status Interrupt, all that is necessary is to send a Reset External/Status Interrupts command. However, if you wish to find the specific cause of the interrupt, it is necessary to read register RR0. In this case, the complete processing takes the following form:

<table>
<thead>
<tr>
<th>Bits sent and received</th>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/D</td>
<td>1 D7 D6 D5 D4 D3 D2 D1 D0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 0 0 0 1 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Effects and Comments

Read register RR0; bit D7 (Break), D5 (Clear To Send), or D3 (Data Carrier Detect) will have had a transition to indicate the cause of the interrupt.

Give a Reset External/Status Interrupts command to set the latches in RR0 to their current values and stop External/Status Interrupts until another transition occurs.

The final kind of interrupt is a Tx Buffer Empty interrupt. If another character is ready to be transmitted on this channel, a Tx Buffer Empty interrupt indicates that it is time to do so. To respond to this interrupt, you need only send the next character. If no other character is ready to transmit, it may be desirable to mark the availability of the transmit mechanism for future use. In addition, you should send a Reset Tx Interrupt Pending command. This command prevents further transmitter interrupts until the next character has been loaded into the transmitter buffer.

The Reset Tx Interrupt Pending command to WR0 takes the following form:

<table>
<thead>
<tr>
<th>Bits sent and received</th>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/D</td>
<td>0 0 1 0 1 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reset Tx Interrupt Pending command; no Tx Empty Interrupts will be given until after the next character has been placed in the transmit buffer.
Z80 Assembler Code

To take these examples further, let us use Z80 Assembler code to implement the routines for a single channel. We assume that the location stored in register WR2 points to the appropriate interrupt service routine. We also assume that the following constants have already been defined:

**SIOctrl.** The address of the SIO's Channel B control port (we assume Channel B in order to include code to initialize the interrupt vector).

**SIOdata.** The address of the SIO's Channel B data port.

**X.** An address pointing to locations in memory that will be used to store various values.

We will write data as binary constants; the "B" suffix indicates this. In most cases, binary constants will be referred to by the command names. We begin with the initialization routine:

```
; place the address of the SIO in the C register for use in subsequent output
LD C, SIOctrl

; load Channel Reset command in A register
LD A, 0001000B
OUT (C), A

; give Channel Reset command
LD A, 0001010B
OUT (C), A

; output basic I/O parameters to WR4
OUT (C), A

; write to register WR0 pointing it to register WR4
LD A, 0001001B
OUT (C), A

; output receive parameters to WR3
LD A, 1110001B
OUT (C), A

; write to register WR0 pointing it to register WR3
LD A, 0001011B
OUT (C), A

; output transmit parameters to WR5
LD A, 1110101B
OUT (C), A

; write to register WR0 pointing it to register WR5
LD A, 0001010B
OUT (C), A

; output transmit parameters to WR5
LD A, 1110101B
OUT (C), A

; output the interrupt vector to WR2; in this case it is decimal location 224
OUT (C), A

; write to register WR0 pointing it to register WR2
LD A, 0001000B
OUT (C), A

; output interrupt parameters to WR1
LD A, 0001001B
OUT (C), A

; return from initialization routine
RET
```

Now let us look first at some sample codes for the Special Receive Condition interrupt routine, following the example above.

```
; save registers which will be used in this routine
PUSH AF

; write to register WR0 pointing it to register RR1
LD A, 00000001B
OUT (SIOctrl), A

; fetch register RR1
IN A, (SIOctrl)

; store result for later error analysis
LD (X), A

; send an Error Reset command to reset device
; latches
LD A, 0011000B
OUT (SIOctrl), A

; fetch the character received—we will discard this
; character since an error occurred during its
; reception
IN A, (SIOdata)

; restore saved registers
POP AF

; enable interrupts
EI

; return from interrupt
RETI
```
Of course, this last routine is probably far too simple to be useful. It is more likely that an interrupt routine will fill up a buffer of characters. A more complex example of a receive interrupt routine is contained in the chapter entitled "A Longer Example."

We now give a simple interrupt routine for an External/Status Interrupt, again assuming that the status contents of SIO register RR0 are stored in temporary location X:

Finally, we give the processing for a transmit interrupt routine in the case where no more characters are to be transmitted.

It is likely that this code would just be a portion of a more general transmit interrupt routine which would transmit a buffer-full of information at a time. A more complex example is included in the section entitled "A Longer Example."

<table>
<thead>
<tr>
<th>SIOrecnt:</th>
<th>PUSH AF</th>
<th>;save registers which will be used in this routine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IN A,(SIOdata)</td>
<td>;fetch the character received</td>
</tr>
<tr>
<td></td>
<td>LD (X),A</td>
<td>;store result for later use</td>
</tr>
<tr>
<td></td>
<td>POP AF</td>
<td>;restore saved registers</td>
</tr>
<tr>
<td></td>
<td>EI</td>
<td>;enable interrupts</td>
</tr>
<tr>
<td></td>
<td>RETI</td>
<td>;return from interrupt</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SIOextmt:</th>
<th>PUSH AF</th>
<th>;save registers which will be used in this routine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LD A,0001000B</td>
<td>;send a Reset External/Status Interrupts command</td>
</tr>
<tr>
<td></td>
<td>OUT (SIOctrl),A</td>
<td>;fetch register RR0</td>
</tr>
<tr>
<td></td>
<td>IN A,(SIOctrl)</td>
<td>;store result for later analysis</td>
</tr>
<tr>
<td></td>
<td>LD (X),A</td>
<td>;restore saved registers</td>
</tr>
<tr>
<td></td>
<td>POP AF</td>
<td>;enable interrupts</td>
</tr>
<tr>
<td></td>
<td>EI</td>
<td>;return from interrupt</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SIOtrmnt:</th>
<th>PUSH AF</th>
<th>;save registers which will be used in this routine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LD A,0010100B</td>
<td>;send a Reset Tx Interrupt Pending command</td>
</tr>
<tr>
<td></td>
<td>OUT (SIOctrl),A</td>
<td>;restore saved registers</td>
</tr>
<tr>
<td></td>
<td>POP AF</td>
<td>;Enable Interrupts</td>
</tr>
<tr>
<td></td>
<td>EI</td>
<td>;Return From Interrupt</td>
</tr>
</tbody>
</table>
**SECTION 5**

**Questions and Answers.**

**Hardware Considerations**

<table>
<thead>
<tr>
<th>Q:</th>
<th>Can a sloppy system clock cause problems in SIO operation?</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:</td>
<td>Yes; the specifications for the system clock are very tight and must be met closely to prevent SIO malfunction. The clock high voltage must be greater than ( V_{CC} - 0.6 \text{V} ) but less than (+5.5\text{V}). The clock low voltage must be greater than (-0.3\text{V}) but less than (+0.45\text{V}). The transitions between these two levels must be made in less than (30\ \text{ns}). This does not apply to the RxC and TxC inputs which are standard TTL levels.</td>
</tr>
</tbody>
</table>

**Register Contents**

<table>
<thead>
<tr>
<th>Q:</th>
<th>When is a received character available to be read?</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:</td>
<td>Data will be available a maximum of (13) system clock cycles from the rising edge of the RxC signal which samples the last bit of the data.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q:</th>
<th>What is the maximum time between character-insertion for transmission and next-character transmission?</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:</td>
<td>This will vary depending on the speed of the line over which the character is being transmitted.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q:</th>
<th>Are the control lines to the SIO synchronous with the system clock so that noise may exist on the buses any time before setup requirements are satisfied?</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:</td>
<td>Yes.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q:</th>
<th>What pins are edge sensitive and should be strapped to avoid strange interrupts?</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:</td>
<td>The external synchronization (SYNC) pins and any other external status pins that are not used, including CTS, and DCD.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q:</th>
<th>What happens if the transmitter or receiver is disabled, while processing a character, by turning off its associated enable bit (bit 3 in register WR5 for transmit or bit 0 in register WR3 for receive)?</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:</td>
<td>The transmitter will complete the character transmission in an orderly fashion. The receiver, however, will not finish. It will lose the character being received and no interrupt will occur.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q:</th>
<th>Does the Tx Buffer Empty (bit 2 in register RR0) get set when the last byte in the buffer is in the process of being shifted out?</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:</td>
<td>No. The bit is set when the transmit buffer has already become empty. Similarly, the Tx Buffer Empty interrupt will not occur until the buffer is empty. The same is true for reception: the Rx Character Available bit (bit 0 in register RR0) is not set until the entire character is in the receive buffer, and the Rx Character Available interrupt will not occur until the entire character has been moved into the buffer.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q:</th>
<th>If an Rx Overrun error occurs (and bit 5 of register RR1 becomes latched on) because a new character has arrived, which character gets lost?</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:</td>
<td>The most recently received character overwrites the next most recently received character.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Q:</th>
<th>Does the Reset External/Status Interrupts command reset any of the status bits in register RR0?</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:</td>
<td>No. However, when a transition occurs on any of the five External/Status bits in register RR0, all of the status bits are latched in their current position until a Reset External/Status Interrupts command is issued. Thus, the command does permit the appropriate bits of register RR0 to reflect the current signal values and should be done immediately after processing each transition on the channel.</td>
</tr>
</tbody>
</table>
Q: If the CPU does not have the return from interrupt sequence (RETI instruction on the Z80 CPU), how may the SIO be informed of the completion of interrupt handling?
A: This may be done by writing the Return From Interrupt command (binary, 00111000) to WR0 in Channel A of the SIO.

Q: If the CPU can be interrupted but cannot be used with vectored interrupts, how should processing be done?
A: Immediately after being interrupted, proceed in a manner similar to polling the SIO for both receive and transmit. Alternatively, the Status Affects Vector bit (bit 2 in register WR1) may be set and a 0 byte placed into the interrupt vector (register WR2 in Channel B). Then, the contents of the interrupt vector can be used to determine the cause of the interrupt and the channel on which the interrupt occurred. This can be queried by reading register RR1 of Channel B. Also, MI should be tied High and no equivalent to an interrupt acknowledge should be issued.

Q: How can the Wait/Ready (W/RDY) signal be used by the CPU in asynchronous I/O?
A: The W/RDY signal is most commonly used in Block Transfer Mode with a DMA, and this use is described in the Z80 DMA Technical Manual. However, W/RDY may be directly connected to the Z80 CPU WAIT line in order to use the block I/O instructions OTDR, OTIR, INDR, and INIR. In this case, the SIO can be used for block transfer reception. To do this, the SIO is configured to interrupt on the first character received only (by settings bits 4 and 3 of register WR1 to 01) and additional characters are sensed using the W/RDY line. The block I/O instructions decrement a byte counter to determine when I/O is complete.

Q: How can the SIO be used to transmit characters containing fewer than 5 bits?
A: First, set bits 6 and 5 in register WR5 to indicate that five or fewer bits per character will be transmitted. The SIO then determines the number of bits to actually transmit from the data byte itself. The data byte should consist of zero or more 1s, three 0s, and the data to be transmitted. Thus, beginning the data byte with 11110001 will cause only the last bit to be transmitted:

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>d</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>d</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>5</td>
</tr>
</tbody>
</table>

*The rightmost number of bits indicated will be transmitted.

Q: Can a Break sequence be sent for a fixed number of character periods?
A: Yes. Break is continuously transmitted as logic 0 by setting bit 4 of register WR5. You can then send characters to the transmitter as long as the Break level is desired to persist. A Break signal, rather than the characters sent, will actually be transmitted, but each bit of each character sent will be clocked as if it were transmitted. The All Sent bit, bit 0 of register RR1, is set to 1 when the last bit of a character is clocked for transmission, and this may be used to determine when to reset bit 4 of register WR5 and stop the Break signal.

Q: If a Break sequence is initiated by setting bit 4 of register WR5, will any character in the process of being transmitted be completed?
A: No. Break is effective immediately when bit 4 of WR5 is set. The “all sent” bit in register RR1 should be monitored to determine when it is safe to initiate a Break sequence.
A Longer Example.

In this section, we give a longer example of asynchronous interrupt-driven full-duplex I/O using the S10. The code for this example is contained in Appendix A, and the basic routines are flowcharted in Figures 7-12.

The example includes code for initialization of the S10, initialization of a receive buffer interrupt routine, and a transfer routine which causes a buffer of up to 80 characters of information to be transmitted on Channel A and a buffer of up to 80 characters of information to be received from Channel A. The transfer routine stops when either all data is received or an error occurs. Completion of an operation on a buffer for both receive and transmit is indicated by a carriage return character. Additional routines (not included in this example) would be needed to call the initialization code and initiate the transfer routine. Therefore, we do not present a complete example; that would only be possible when all details of a particular communication environment and operating system were known.

The code begins by defining the value of the S10 control and data channels, followed by location definitions for the interrupt vector. There is then a series of constant definitions of the various fields in each register of the S10. This is followed by a table-driven S10 initialization routine called "S10_init," shown in Figure 7, which uses the table beginning at the location "SIOtable." The SIO_init routine initializes the S10 with exactly the same
A Longer Example
(Continued)

parameters as the interrupt-driven example in the previous section. The table-driven version
is presented simply as an alternative means of coding this material.

A short routine for filling the receive buffer with "FF" (hex) characters and buffer definitions follows the SIO_Init routine. This in turn is followed by the transfer routine, Figure 8, which begins transmitting on Channel A; transmission and reception is thereafter directed by the interrupt routines. After the transfer routine begins output, it checks for
various error conditions and loops until there is either completion or an error.

Then the four interrupt routines follow: TxBEmpty, Figure 9, is called on a transmit buffer interrupt; it begins transmission of the next character in the buffer. A carriage return stops transmission. RecvChar, Figure 10, is called on a normal receive interrupt; it places the received character in the buffer if the buffer is not full and updates receive counters. The routines SpRecvChar, Figure 11, and ExtStatus, Figure 12, are error interrupts; they update information to indicate the nature of the error.

The code of this example can be used in a situation where data is being sent to a device which echoes the data sent. In such a case, the transmit and receive buffers could be compared upon completion for line or transmission errors.

Figure 10. Receive Character Interrupt Routine

Figure 11. Special Receive Condition Interrupt Routine

Figure 12. External/Status Interrupt Routine
Appendix A

Interrupt-Driven Code Example

Table of Interrupt Vectors

The table (Int_Tab) starts at the lowest priority vector, which should be dddd0000d

<table>
<thead>
<tr>
<th>SIO Port Identifiers and System Address Bus Addresses</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIO: EQU 40H</td>
</tr>
<tr>
<td>SIOData: EQU SIO + 1</td>
</tr>
<tr>
<td>SIOCtrl: EQU SIO + 2</td>
</tr>
<tr>
<td>SIOBData: EQU SIO + 3</td>
</tr>
<tr>
<td>SIOBCtrl: EQU SIO + 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command Identifiers and Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Includes all control bytes for asynchronous and synchronous I/O.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WR0 Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0: EQU 00H, SIO register pointers</td>
</tr>
<tr>
<td>R1: EQU 01H</td>
</tr>
<tr>
<td>R2: EQU 02H</td>
</tr>
<tr>
<td>R3: EQU 03H</td>
</tr>
<tr>
<td>R4: EQU 04H</td>
</tr>
<tr>
<td>R5: EQU 05H</td>
</tr>
<tr>
<td>R6: EQU 06H</td>
</tr>
<tr>
<td>R7: EQU 07H</td>
</tr>
<tr>
<td>NC: EQU 00H, Null Code</td>
</tr>
<tr>
<td>SA: EQU 08H, Send Abort (SDLC)</td>
</tr>
<tr>
<td>RESI: EQU 10H, Reset ExtStat Int</td>
</tr>
<tr>
<td>CHRST: EQU 16H, Channel Reset</td>
</tr>
<tr>
<td>EIONRC: EQU 20H, Enable Int On Next Rx Char</td>
</tr>
<tr>
<td>RTP: EQU 28H, Reset Tx Int Pending</td>
</tr>
<tr>
<td>ER: EQU 30H, Error Reset</td>
</tr>
<tr>
<td>RFI: EQU 38H, Return From Int</td>
</tr>
<tr>
<td>RRCC: EQU 40H, Reset Rx CRC Checker</td>
</tr>
<tr>
<td>RTCG: EQU 80H, Reset Tx CRC Generator</td>
</tr>
<tr>
<td>RTUEL: EQU 00H, Reset Tx Under/EOM Latch</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIO_Init: LD HL, Int_Tab</td>
</tr>
<tr>
<td>INC HL</td>
</tr>
<tr>
<td>CP 0</td>
</tr>
<tr>
<td>RET 2</td>
</tr>
<tr>
<td>OUT (SIOACtrl), A</td>
</tr>
<tr>
<td>JSR Int_Init, loop for initialization</td>
</tr>
<tr>
<td>SIOtable: DEFB CR, table for initialization</td>
</tr>
<tr>
<td>DEFB B4 + RESI</td>
</tr>
<tr>
<td>DEFB C64 + ODD + PARITY + S2</td>
</tr>
<tr>
<td>DEFB R3 + RESI</td>
</tr>
<tr>
<td>DEFB B8 + AUTOEN + ENRCVR</td>
</tr>
<tr>
<td>DEFB R5 + RESI</td>
</tr>
<tr>
<td>DEFB DTR + RTS + T8 + XENABL</td>
</tr>
<tr>
<td>DEFB R2 + RESI</td>
</tr>
<tr>
<td>L_Loc: DEFS 1, location of int table</td>
</tr>
<tr>
<td>DEFB R1 + RESI, address</td>
</tr>
<tr>
<td>DEFB EXTIE + XMRIE + SAVECT + PAVECT</td>
</tr>
<tr>
<td>DEFB 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WR1 Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>WAIT: EQU 00H, Wait function</td>
</tr>
<tr>
<td>DRCVRI: EQU 00H, Disable Receive interrupts</td>
</tr>
<tr>
<td>EXTE: EQU 01H, External interrupt enable</td>
</tr>
<tr>
<td>XMRRIE: EQU 02H, Transmit interrupt enable</td>
</tr>
<tr>
<td>SAVECT: EQU 04H, Status affects vector</td>
</tr>
<tr>
<td>FIRSTC: EQU 06H, Rx interrupt on first character</td>
</tr>
<tr>
<td>PAVECT: EQU 10H, Rx interrupt on all characters</td>
</tr>
<tr>
<td>PDAVECT: EQU 18H, Rx interrupt on all characters</td>
</tr>
<tr>
<td>WRONRT: EQU 20H, Wait/Ready on receive</td>
</tr>
<tr>
<td>RRY: EQU 40H, Ready function</td>
</tr>
<tr>
<td>WRDYEN: EQU 80H, Wait/Ready enable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WR2 Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>IV: EQU 00H</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WR3 Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>B5: EQU 00H, Receive 5 bits/character</td>
</tr>
<tr>
<td>RENABL: EQU 01H, Receiver enable</td>
</tr>
<tr>
<td>ENRCVR: EQU 01H, Receiver enable</td>
</tr>
<tr>
<td>SCLINH: EQU 02H, Sync character load inhibit</td>
</tr>
<tr>
<td>ADSRCH: EQU 04H, Address search mode</td>
</tr>
<tr>
<td>RORCEN: EQU 08H, Receive CRC enable</td>
</tr>
<tr>
<td>HUNT: EQU 10H, Enter hunt mode</td>
</tr>
<tr>
<td>AUTOEN: EQU 20H, Auto enables</td>
</tr>
<tr>
<td>B7: EQU 40H, Receive 7 bits/character</td>
</tr>
<tr>
<td>B6: EQU 80H, Receive 6 bits/character</td>
</tr>
<tr>
<td>B8: EQU 00H, Receive 8 bits/character</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WR4 Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYNC: EQU 00H, Sync modes enable</td>
</tr>
<tr>
<td>NOPRTY: EQU 00H, Disable parity</td>
</tr>
<tr>
<td>ODD: EQU 00H, Odd parity</td>
</tr>
<tr>
<td>MONO: EQU 00H, 8 bit sync character</td>
</tr>
<tr>
<td>C1: EQU 00H, X1 clock mode</td>
</tr>
<tr>
<td>PARITY: EQU 01H, Enable parity</td>
</tr>
<tr>
<td>EVEN: EQU 02H, Even parity</td>
</tr>
<tr>
<td>S1: EQU 04H, 1 stop bit/character</td>
</tr>
<tr>
<td>S1/HALF: EQU 08H, 1 and a half stop bits/character</td>
</tr>
<tr>
<td>S2: EQU 0CH, 2 stop bits/character</td>
</tr>
<tr>
<td>BISYNC: EQU 10H, 16 bit sync character</td>
</tr>
<tr>
<td>SDLX: EQU 20H, SDLX mode</td>
</tr>
<tr>
<td>ENSYNC: EQU 30H, External sync mode</td>
</tr>
<tr>
<td>C16: EQU 40H, CX16 clock mode</td>
</tr>
<tr>
<td>C32: EQU 80H, X32 clock mode</td>
</tr>
<tr>
<td>C64: EQU 00H, X64 clock mode</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>WR5 Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS: EQU 00H, Transmit 5 bits/character</td>
</tr>
<tr>
<td>XCRCN: EQU 01H, Transmit CRC enable</td>
</tr>
<tr>
<td>RTS: EQU 02H, Request to send</td>
</tr>
<tr>
<td>SELCRC: EQU 04H, Select CRC-16 polynomial</td>
</tr>
<tr>
<td>XENAB: EQU 08H, Transmitter enable</td>
</tr>
<tr>
<td>BREAK: EQU 10H, Send break</td>
</tr>
<tr>
<td>T7: EQU 20H, Transmit 7 bits/character</td>
</tr>
<tr>
<td>T6: EQU 40H, Transmit 6 bits/character</td>
</tr>
<tr>
<td>T8: EQU 60H, Transmit 8 bits/character</td>
</tr>
<tr>
<td>DTR: EQU 80H, Data terminal ready</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Initialization</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIO_Init: LD HL, Int_Tab</td>
</tr>
<tr>
<td>INC HL</td>
</tr>
<tr>
<td>CP 0</td>
</tr>
<tr>
<td>RET 2</td>
</tr>
<tr>
<td>OUT (SIOACtrl), A</td>
</tr>
<tr>
<td>JSR Int_Init, loop for initialization</td>
</tr>
<tr>
<td>SIOtable: DEFB CR, table for initialization</td>
</tr>
<tr>
<td>DEFB B4 + RESI</td>
</tr>
<tr>
<td>DEFB C64 + ODD + PARITY + S2</td>
</tr>
<tr>
<td>DEFB R3 + RESI</td>
</tr>
<tr>
<td>DEFB B8 + AUTOEN + ENRCVR</td>
</tr>
<tr>
<td>DEFB R5 + RESI</td>
</tr>
<tr>
<td>DEFB DTR + RTS + T8 + XENABL</td>
</tr>
<tr>
<td>DEFB R2 + RESI</td>
</tr>
<tr>
<td>L_Loc: DEFS 1, location of int table</td>
</tr>
<tr>
<td>DEFB R1 + RESI, address</td>
</tr>
<tr>
<td>DEFB EXTIE + XMRIE + SAVECT + PAVECT</td>
</tr>
<tr>
<td>DEFB 0</td>
</tr>
</tbody>
</table>
Receiver Buffer Initialization

Bu_Imt. LD A,BufLength .fill receiver buffer
LD B,A , with FF characters
LD HL,RBuffer . to detect errors
LD A,OFFH

Bu_l LD (HL),A ,a loop for Bu_Imt
INC HL LD B,A
DJNZ Bu_l
LD A,(RBufCtr)
RET

BufLength: EQU 80 ;buffer length
XBuffer DEFS BufLength .Tx buffer starting location
RBuPtr. DEFS BufLength .Rx buffer starting location
XBuPtr DEFS 2 .Rx pointer
RBuPtr DEFS 2 .Rx pointer
RBuCtr. DEFS 1 .Rx counter

Transmit Routine (see Figure 8)
Initiates transmission of a buffer-full of data and terminates when
an error is detected or a complete buffer has been received
RxStat DEFS 1 ,Receive Status Word
TxStat DEFS 1 ,Transmit Status Word
Complete EQU 1
CR: EQU 0DH
Break EQU 80H
BOM EQU 80H
Overflow EQU 0FFH

Transfer LD HL,XBuffer .setup to begin Tx
INC HL
LD (XBuPtr),HL
LD HL,RBuffer
LD (RBuPtr),HL
XOR A ,A = 0
LD (RBuCtr),A
LD (TxStat),A
LD (RxStat),A
LD A,SIOData ;start Tx task
LD C,A
LD HL,(XBuffer) ,first character
LD A,(HL)
OUT (C),A

Tloop. LD A,(TxStat) ,await Tx completion or error
CP 0
RET NZ
LD A,(RxStat)
CP Overflow
RET Z
CP Complete
RET Z
JR NZ,Tloop
RET

Transmitter Buffer Empty Routine (see Figure 9)

TxBEmpty PUSH AF
PUSH BC
PUSH HL
LD HL,(XBuPtr)
LD A,SIOData
LD C,A
LD A,(HL)
OUT (C),A
CP CR
JR NZ,TxBExit ,last character?
LD A,RTIP ,Reset Tx Int Pending
INC C
OUT (C),A ;to control port

TxBExit LD (XBuPtr),HL ;save pointer
POP HL
POP BC
POP AF
EI
RETI

Receive Character Routine (see Figure 10)

RxChar. PUSH AF
PUSH BC
LD A,SIOData
LD C,A
IN A,(C) ,get character
LD B,A
LD A,(RBuCtr)
CP BuflLength
JR Z,Over
INC A ;bump counter
LD (RBuCtr),A
LD A,B
LD HL,(RBuPtr) ,bump pointer
LD (HL),A
INC HL
LD (RBuPtr),HL
CP CR
JR Z,Over
LD A,Complete
LD (RxStat),A
JR RxExit

Over LD A,Overflow ,indicate error
LD (RxStat),A

RxExit POP BC
POP AF
EI
RETI

Special Receive Condition Routine (see Figure 11)

SpRxCond: PUSH AF
PUSH BC
LD A,SIOData
LD C,A
LD A,RI ,get RR1
INC C
OUT (C),A
IN A,(C)
LD (RxStat),A ,save status
LD A,ER ;Reset Errors
DEC C
OUT (C),A
DEC C
IN A,(C) ;get character
POP BC
POP AF
EI
RETI

External/Status Routine (see Figure 12)

ExtStatus: PUSH AF
PUSH BC
LD A,SIOCtrl
LD C,A
IN A,(C) ;get RR0
LD (TxStat),A
LD A,RESI ,Reset Ext Stat Int
OUT (C),A
POP BC
POP AF
EI
RETI

END
Appendix B
Read Register Bit Functions

READ REGISTER 0

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rx CHARACTER AVAILABLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT PENDING (CH A ONLY)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tx BUFFER EMPTY</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DCD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SYNCH/UN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CTS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rx UNDERRUN/EOM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BREAK/ABORT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Used With 'External/Status Interrupt' Mode

READ REGISTER 1

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL SENT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 FIELD BITS IN PREVIOUS BYTE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 0 0 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 0 0 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 1 0 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0 1 0 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 1 0 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1 1 0 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 2 8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Residue Data For Eight Rx Bits/Character Programmed

READ REGISTER 2

<table>
<thead>
<tr>
<th>D7</th>
<th>D6</th>
<th>D5</th>
<th>D4</th>
<th>D3</th>
<th>D2</th>
<th>D1</th>
<th>D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>V8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Variable & Status Affects Vector* Programmed
# Appendix C

## Write Register Bit Functions

### WRITE REGISTER 0

<table>
<thead>
<tr>
<th>D7 D6 D5 D4 D3 D2 D1 D0</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>NULL CODE</td>
</tr>
<tr>
<td>0 0 1</td>
<td>REGISTER 0</td>
</tr>
<tr>
<td>0 1 0</td>
<td>REGISTER 1</td>
</tr>
<tr>
<td>0 1 1</td>
<td>REGISTER 2</td>
</tr>
<tr>
<td>1 0 0</td>
<td>REGISTER 3</td>
</tr>
<tr>
<td>1 0 1</td>
<td>REGISTER 4</td>
</tr>
<tr>
<td>1 1 0</td>
<td>REGISTER 5</td>
</tr>
<tr>
<td>1 1 1</td>
<td>REGISTER 7</td>
</tr>
</tbody>
</table>

- 0 0 0: NULL CODE
- 0 0 1: SEND ABORT (SDLC)
- 0 1 0: RESET EXT/STATUS INTERRUPTS
- 0 1 1: CHANNEL RESET
- 1 0 0: ENABLE INT ON NEXT Rx CHARACTER
- 1 0 1: RESET UINT PENDING
- 1 1 0: ERROR RESET
- 1 1 1: RETURN FROM INT (CH A ONLY)

### WRITE REGISTER 1

<table>
<thead>
<tr>
<th>D7 D6 D5 D4 D3 D2 D1 D0</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>NULL CODE</td>
</tr>
<tr>
<td>0 1 0</td>
<td>RESET Rx CRC CHECKER</td>
</tr>
<tr>
<td>1 0 0</td>
<td>RESET Tx CRC GENERATOR</td>
</tr>
<tr>
<td>1 1 0</td>
<td>RESET Tx UNDERRUN/DOM LATCH</td>
</tr>
</tbody>
</table>

- 0 0 0: NULL CODE
- 0 1 0: Rx INT DISABLE
- 0 1 1: Rx INT ON FIRST CHARACTER
- 1 0 0: INT ON ALL Rx CHARACTER (PARITY AFFECTS VECTOR)
- 1 0 1: INT ON ALL Rx CHARACTERS (PARITY DOES NOT AFFECT VECTOR)
- 1 1 0: RETURN FROM INT (CH A ONLY)

### WRITE REGISTER 2 (CHANNEL B ONLY)

<table>
<thead>
<tr>
<th>D7 D6 D5 D4 D3 D2 D1 D0</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>V0</td>
<td>INTERRUPT VECTOR</td>
</tr>
<tr>
<td>V1</td>
<td>V2</td>
</tr>
<tr>
<td>V3</td>
<td>V4</td>
</tr>
<tr>
<td>V5</td>
<td>V6</td>
</tr>
<tr>
<td>V7</td>
<td></td>
</tr>
</tbody>
</table>

### WRITE REGISTER 3

<table>
<thead>
<tr>
<th>D7 D6 D5 D4 D3 D2 D1 D0</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>Rx 5 BITS/CHARACTER</td>
</tr>
<tr>
<td>0 1 0</td>
<td>Rx 7 BITS/CHARACTER</td>
</tr>
<tr>
<td>1 0 0</td>
<td>Rx 8 BITS/CHARACTER</td>
</tr>
<tr>
<td>1 1 0</td>
<td>Auto Enables</td>
</tr>
</tbody>
</table>

### WRITE REGISTER 4

<table>
<thead>
<tr>
<th>D7 D6 D5 D4 D3 D2 D1 D0</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>NULL CODE</td>
</tr>
<tr>
<td>0 0 1</td>
<td>8 BIT SYNC CHARACTER</td>
</tr>
<tr>
<td>0 1 0</td>
<td>16 BIT SYNC CHARACTER</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1) SDLC MODE (0111110) FLAG</td>
</tr>
<tr>
<td>1 1 0</td>
<td>2) EXTERNAL SYNC MODE</td>
</tr>
</tbody>
</table>

- 0 0 0: NULL CODE
- 0 0 1: Rx INT DISABLE
- 0 1 1: Rx INT ON FIRST CHARACTER
- 1 0 0: INT ON ALL Rx CHARACTERS (PARITY AFFECTS VECTOR)
- 1 0 1: INT ON ALL Rx CHARACTERS (PARITY DOES NOT AFFECT VECTOR)
- 1 1 0: RETURN FROM INT (CH A ONLY)

### WRITE REGISTER 5

<table>
<thead>
<tr>
<th>D7 D6 D5 D4 D3 D2 D1 D0</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>NULL CODE</td>
</tr>
<tr>
<td>0 0 1</td>
<td>Tx 5 BITS OR LESS/CHARACTER</td>
</tr>
<tr>
<td>0 1 0</td>
<td>Tx 7 BITS/CHARACTER</td>
</tr>
<tr>
<td>1 0 1</td>
<td>Tx 8 BITS/CHARACTER</td>
</tr>
<tr>
<td>1 1 0</td>
<td>Tx 8 BITS/CHARACTER</td>
</tr>
</tbody>
</table>

- 0 0 0: NULL CODE
- 0 0 1: Rx INT DISABLE
- 0 1 1: Rx INT ON FIRST CHARACTER
- 1 0 0: INT ON ALL Rx CHARACTERS (PARITY AFFECTS VECTOR)
- 1 0 1: INT ON ALL Rx CHARACTERS (PARITY DOES NOT AFFECT VECTOR)
- 1 1 0: RETURN FROM INT (CH A ONLY)

### WRITE REGISTER 6

<table>
<thead>
<tr>
<th>D7 D6 D5 D4 D3 D2 D1 D0</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>V0</td>
<td>SYNC BIT 0</td>
</tr>
<tr>
<td>V1</td>
<td>SYNC BIT 2</td>
</tr>
<tr>
<td>V2</td>
<td>SYNC BIT 3</td>
</tr>
<tr>
<td>V3</td>
<td>SYNC BIT 4</td>
</tr>
<tr>
<td>V4</td>
<td>SYNC BIT 5</td>
</tr>
<tr>
<td>V5</td>
<td>SYNC BIT 6</td>
</tr>
<tr>
<td>V6</td>
<td>SYNC BIT 7</td>
</tr>
</tbody>
</table>

### WRITE REGISTER 7

<table>
<thead>
<tr>
<th>D7 D6 D5 D4 D3 D2 D1 D0</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>V0</td>
<td>SYNC BIT 8</td>
</tr>
<tr>
<td>V1</td>
<td>SYNC BIT 9</td>
</tr>
<tr>
<td>V2</td>
<td>SYNC BIT 10</td>
</tr>
<tr>
<td>V3</td>
<td>SYNC BIT 11</td>
</tr>
<tr>
<td>V4</td>
<td>SYNC BIT 12</td>
</tr>
<tr>
<td>V5</td>
<td>SYNC BIT 13</td>
</tr>
<tr>
<td>V6</td>
<td>SYNC BIT 14</td>
</tr>
<tr>
<td>V7</td>
<td>SYNC BIT 15</td>
</tr>
</tbody>
</table>

*For SDLC it Must be Programmed to 0111110* For flag recognition.
Using the Z80 SIO With SDLC

Application Brief

March 1981

The reader should be familiar with hardware aspects of the SIO such as interfacing to the CPU and a modem. A more detailed description of the SDLC protocol is given in the IBM publication Synchronous Data Link Control General Information (document # GA27-3093-2). A description of the Z80 SIO can be found in the Zilog Data Book (document # 00-2034-A).

INTRODUCTION

This application brief describes the use of the Z80 SIO with the increasingly popular Synchronous Data Link Control (SDLC) communications protocol. A general description of the SDLC protocol and implementation of the protocol using the SIO are discussed. Descriptions for transmit and receive operations are given for use with simple control frame sequences.

DESCRIPTION

Data communication today requires a communication protocol that can transfer data quickly and reliably. One such protocol, Synchronous Data Link Control (SDLC), is the link control used by the IBM Systems Network Architecture (SNA) communication package. SDLC is actually a subset of the International Standards Organization (ISO) link control called High Level Data Link Control (HDLC), which is used for international data communication.

SDLC is a Bit-Oriented Protocol (BOP). It differs from Byte-Control Protocols (BCPs), such as bisync, in having a few bit patterns for control functions instead of several special character sequences. The attributes of the SDLC protocol are position dependent rather than character dependent, so control is determined by the location of the byte as well as by the bit pattern.

A character in SDLC is sent as an octet, a group of eight bits. Several octets combine to form a message frame in such a way that each octet belongs to a particular field. Each message frame consists of an opening flag, address, control, information, Frame Check Sequence (FCS), and closing flag fields. The flag field contains a unique binary pattern, 01111110, which indicates the beginning and end of a message frame. This pattern simplifies the hardware interface in receiving devices so that multiple devices connected to a common link do not conflict with one another. The receiving devices respond only after a valid flag character has been detected. Once communication is established for a particular device, the other devices ignore the message until the next flag character is detected.

The address field contains one or more octets that are used to select a particular station on the data link. An address of all Is is a global address code that selects all the devices on the link. When a primary station sends a frame, the address field is used to select a secondary station. When a secondary station sends a message to the primary station, the address field contains the secondary station address, i.e., the source of the message.

The control field follows the address field and contains information about the type of frame being sent. The control field consists of one octet and is always present.

The information field consists of zero or more 8-bit octets and contains any actual data transferred. However, because of the limitations of the error-checking algorithm used in the frame-check sequence, maximum recommended block size is approximately 4096 octets.

The Frame Check Sequence (FCS) follows the information field or the control field, depending on the type of message frame sent. The FCS is a 16-bit Cyclic Redundancy Code (CRC) of the bits in the address, control, and information fields. The FCS is based on the CRC-CCITT code, which uses the polynomial $x^{16} + x^{12} + x^5 + 1$. The Z80 SIO contains the circuitry necessary to generate and check the FCS field.
Zero Insertion/deletion is a feature of SDLC that allows any data pattern to be sent. Zero insertion occurs when five consecutive 1s in the data pattern are transmitted. After the fifth 1, a 0 is inserted before the next bit is sent. The data is not affected in any way except that there is an extra 0 in the data stream. The receiver counts the 1s and deletes the 0 following the five consecutive 1s, thus restoring the original data pattern. Zero Insertion and deletion is necessary because of the hardware constraint of searching for a flag character or abort sequence. Six 1s preceded and followed by a 0 indicate a flag character. Seven to 14 1s signify an abort, while an idle line (inactive) is indicated by 15 or more 1s. Under these three conditions, zero insertion/deletion is inhibited. Figure 2 illustrates the various line conditions.

SDLC protocol differs from other synchronous protocols with respect to frame timing. In bsync, for example, a host computer might interrupt transmission temporarily by sending sync characters instead of data. This suspended condition could continue as long as the receiver does not time out. With SDLC, however, it is illegal to send flags in the middle of a frame to idle the line. Such an occurrence causes an error condition and disrupts orderly operation. Therefore, the transmitting device must send a complete frame without interruption. If a message cannot be completed, the primary station sends an abort and resumes message transmission later. These conditions are discussed later in the Programming section of this brief.

### Figure 1. A Typical SDLC Message Frame Format

<table>
<thead>
<tr>
<th>Flag (Beginning of message frame)</th>
<th>Address</th>
<th>Control</th>
<th>Information</th>
<th>FCS</th>
<th>Flag (End of message frame)</th>
</tr>
</thead>
<tbody>
<tr>
<td>01111110 8-bit character</td>
<td>One</td>
<td>One</td>
<td>Zero or more 8-bit characters</td>
<td>16-bit CRC-CCITT</td>
<td>01111110</td>
</tr>
</tbody>
</table>

### Figure 2. Bit Patterns for Various Line Conditions

- **a) Zero Insertion**
  - Address = 10110000
  - Control = 01111111
  - Zero Insertion

- **b) Abort Condition**
  - XXXX111111101111110...
  - Abort Flag

- **c) Idle Condition**
  - XXXX1111111111111...
  - Idle
Implementation of the SDLC protocol with the Z80 SIO is simplified by the design of the SIO. This section discusses four areas of SIO programming: initialization, transmit operation, receive operation, and exception condition processing.

Initialization defines the basic mode of operation for the SIO. Table 1 shows the sequence of steps used to initialize the SIO, along with the necessary parameters. Since vectored interrupts are used, the SIO is programmed with the status affects vector (SAV) bit (WR1, bit 2) set.

Other function bits that can be included are the external interrupt enable bit (WR1, bit 0), which results in an interrupt for each DCD or CTS change, Tx underrun or abort change; address search bit (WR3, bit 2), which when set, prevents the SIO from responding to data received unless the address byte matches the contents of WR6 or the global (FFH) address; auto enable bit (WR3, bit 5), which causes the inactive CTS level to disable the transmitter and the inactive DCD level to disable the receiver; and DTR (WR5, bit 7) and RTS (WR5, bit 1), which can be used to control a modem or other such device.

After the SIO has been initialized and enabled, it can begin sending SDLC frames by software activation of the transmitter. Activating the transmitter includes resetting the transmitter inactive semaphore (a program indicator), resetting the TX CRC accumulation, sending a character to the SIO, and resetting the TX underrun/EOM latch in the SIO. Figure 3 shows the sequence for transmitting a typical control message frame using interrupts.

Once the SIO is initialized and the transmitter is enabled, it sends flag characters continuously until a message begins transmission. These flag characters consist of the full 8-bit pattern. Although the SIO can receive flag characters with shared Os (0111110111110111110...), it can only transmit flag characters without shared Os (011111100111110011111...).

### Table 1. SIO Initialization Sequence

<table>
<thead>
<tr>
<th>Register</th>
<th>Data</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00011000</td>
<td>Channel reset</td>
</tr>
<tr>
<td>2</td>
<td>(Vector)</td>
<td>Interrupt vector lower eight bits (channel B only)</td>
</tr>
<tr>
<td>4</td>
<td>00100000</td>
<td>SDLC mode</td>
</tr>
<tr>
<td>1</td>
<td>00011111</td>
<td>Interrupt control</td>
</tr>
<tr>
<td>6</td>
<td>(Address)</td>
<td>Rx address field</td>
</tr>
<tr>
<td>7</td>
<td>01111110</td>
<td>Flag field</td>
</tr>
<tr>
<td>5</td>
<td>11101011</td>
<td>Tx character length, enable, CRC enable RTS and DTR</td>
</tr>
<tr>
<td>3</td>
<td>11001001</td>
<td>Rx character length, enable, and CRC enable</td>
</tr>
</tbody>
</table>

### Figure 3. A Typical Transmit Control Frame Sequence
When the SIO is loaded with the first data character (address byte), it stores the character in the Tx buffer until the current flag character has completed shifting. After the address byte is transferred into the shift register, a Transmit Buffer Empty (TBE) interrupt occurs. The program then loads the control character into the SIO and continues processing. The next TBE interrupt is ignored by the program (and no further data is sent to the SIO), but a Reset Tx Interrupt Pending command is issued to the SIO to clear the TBE interrupt condition. Also, the program Message completed (MC) semaphore is set so that appropriate action can be taken when the next TBE interrupt occurs.

When the last data character (the control byte) has been shifted out of the SIO, the Tx underrun/EOl latch is set because the SIO buffer was not loaded with a character on the previous TBE interrupt. As a result, an External/Status Change (ESC) interrupt occurs and the SIO begins transmitting the FCS bytes automatically. In the ESC interrupt service routine, the program checks for other condition changes including CTS, DCD, and abort, and passes the status on to the program at the next-higher level.

After the FCS bytes have been shifted out, the SIO generates a TBE interrupt to indicate that a flag character is being transmitted. The TBE interrupt service routine interprets the MC semaphore and determines that the frame has completed transmission. The program then clears the MC semaphore, sets the Transmitter inactive semaphore, starts a timer for a response from the receiving device, and clears the TBE interrupt condition. At this point, transmission of an SDLC message frame is complete and another message frame may be sent.

If the transmitter is to be turned off, the program must allow at least a two-character time delay before disabling the transmitter. This can be accomplished by connecting the SIO Tx clock line to the input of a counter and having the counter interrupt the CPU when the bit count expires.

The SDLC receive sequence is slightly less complex than the transmit sequence. To begin, the SIO enters Hunt mode when any of three conditions occurs: receive enable, abort detect, or a software command. In Hunt mode the SIO searches for flag characters, and when it detects a flag, the SIO generates an ESC Interrupt. This interrupt can be used to signal line activation or the end of an abort condition, depending upon the previous receive condition. For example, when the SIO has been initialized, the receive circuitry is enabled and immediately begins searching for flag characters (Hunt mode operation). When the first flag is detected, the S10 exits from Hunt mode, which results in an ESC interrupt, and the SIO begins searching for the address field. If the SIO is programmed for Address Search mode and an address is received that does not match the programmed address byte in the SIO, the SIO does nothing until the next flag is found, after which the SIO again searches for an address match.

**RECEIVE OPERATION**

The SDLC receive sequence is slightly less complex than the transmit sequence. To begin, the SIO enters Hunt mode when any of three conditions occurs: receive enable, abort detect, or a software command. In Hunt mode the SIO searches for flag characters, and when it detects a flag, the SIO generates an ESC Interrupt. This interrupt can be used to signal line activation or the end of an abort condition, depending upon the previous receive condition. For example, when the SIO has been initialized, the receive circuitry is enabled and immediately begins searching for flag characters (Hunt mode operation). When the first flag is detected, the SIO exits from Hunt mode, which results in an ESC interrupt, and the SIO begins searching for the address field. If the SIO is programmed for Address Search mode and an address is received that does not match the programmed address byte in the SIO, the SIO does nothing until the next flag is found, after which the SIO again searches for an address match.

**SDLC RX**

![Control Message Frame Diagram](image)

**NOTES**

* The SRC routine normally reads the data character to clear the SIO buffer. This should be done after the program issues an Error Reset command.

†RCA = Receive Character Available

++SRC = Special Receive Condition (higher priority than RCA)

Figure 4. A Typical Receive Control Frame Sequence
If the address field matches the address byte programmed into the S10, the S10 generates a Receive Character Available (RCA) interrupt when the address byte is ready to be transferred from the S10 to the CPU. If the S10 is programmed to interrupt on all receive characters, it generates an RCA interrupt for each character received thereafter. It should be noted that the S10 generates the RCA interrupt when a character reaches the top of the receive FIFO rather than when a character is transferred from the shift register to the FIFO. This means that if the FIFO is full of data, each character generates a separate RCA interrupt. This results in a more consistent software routine that does not need to check the receive FIFO, provided there is enough time between character transfers to allow the routine to complete the processing for each character.

After the last FCS byte of a frame is received and processed, the S10 generates a Special Receive Condition (SRC) interrupt, which is of higher priority than the RCA interrupt. In the SRC service routine, RRI is read to determine the cause of the interrupt and the appropriate program semaphores are updated. Normal completion results in no FCS or overrun errors and the End-of-Frame bit is set. Upon completion of the SRC interrupt service routine, the program issues an Error Reset command to the S10 and reads the data port to discard the received data. If the data is not read and discarded, an RCA interrupt occurs. Now, a complete message frame and the first FCS byte are in the receive buffer.

Figure 4 shows the sequence for a typical control frame received by the S10. If the address field byte is to be discarded, a program semaphore should initially be set to signal this to the RCA routine. After the address field has been received, the semaphore is cleared and reception continues normally. Note that upon completion of a frame, an RCA interrupt is generated for the first FCS byte and an SRC interrupt is generated for the last CRC byte.

Table 2 lists the contents of the interrupt service routines used with the S10. The wake routine is not an interrupt service routine but is a routine called by the program on the next higher level to begin frame transmission. Once the wake routine is called, the program on the next higher level monitors the Tx active semaphore to determine when the current frame completes transmission and the next frame transmission can begin.

<table>
<thead>
<tr>
<th>Wake: Clear Tx inactive semaphore</th>
<th>Receive Character Available (RCA): Read SIO RRI</th>
<th>Transmit Buffer Empty (TBE): If (MC cleared)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset Tx CRC</td>
<td>If (EOF) Read and discard data</td>
<td>If (buffer not empty) Data to S10</td>
</tr>
<tr>
<td>Data to S10</td>
<td>Else,</td>
<td>Else,</td>
</tr>
<tr>
<td>(Address field byte)</td>
<td>Set MC semaphore</td>
<td>Clear MC</td>
</tr>
<tr>
<td>Reset Tx Underrun/EOV latch</td>
<td>Reset TBE condition</td>
<td>Set Tx inactive</td>
</tr>
<tr>
<td></td>
<td>Else,</td>
<td>Reset TBE condition</td>
</tr>
<tr>
<td></td>
<td>Clear MC</td>
<td>Start Response timer</td>
</tr>
</tbody>
</table>

Table 2. S10 SDLC Interrupt Service Routines
Most of the exception conditions encountered in the SDLC protocol have been discussed in the previous sections. They include abort detect and DCD or CTS change. This section further describes some of the more unusual conditions.

**DCD and CTS Change.** The program handles DCD and CTS change by updating its semaphores each time an ESC interrupt occurs. In this manner, the program on the next higher level monitors the semaphores and determines a course of action based on what these semaphores indicate.

**Abort and Idle Line Detect.** Abort and idle line detect are a bit more complicated, since they result in similar interrupt operations. An abort occurs during a valid message frame, if the abort time is greater than 14 bits, an idle line is detected. This detection can be done by activating a timer when the ESC interrupt that signals a marking line occurs. If another ESC interrupt occurs before the timer times out, the line is in an abort condition. If the timer times out before another ESC interrupt occurs, then the line is idle and the program can pursue an appropriate course of action. A possible mechanism for implementing the timer function is to use a programmable counter that is tied to the receive clock line to count bits. The counter is programmed for eight clock transitions and is started as soon as the SIO interrupts the CPU with an abort condition. Only eight clock transitions need to be counted because by the time the SIO generates the ESC interrupt, at least seven Is have already passed. Figure 6 shows the abort/idle line timing and the interrupts resulting from the line changes.

![Timetable](image)

**Figure 6. Abort/Idle Line Conditions**

This brief describes implementation of the SDLC protocol using the SIO in an interrupt-driven environment. Descriptions for transmit and receive operations are given for use with simple control frame sequences. For frames that transfer data, the sequences are similar except for transmit, where a data character is sent to the SIO for a TBE interrupt. For receive, multiple RCA interrupts occur for each data byte received.

The Z80 SIO enhances system performance by minimizing CPU intervention during data transfers using the SDLC protocol. Performance can be improved further by using the Z80 DMA with the SIO, resulting in an efficient system configuration that reduces CPU interaction to a minimum.

**APPENDIX**

Following is the listing of a simple SIO test program that uses the SDLC protocol. This program uses vectored interrupts to send a short SDLC control frame consisting of Address 9EH, Control 19H, and Data 81H. The response timer times the response of the receiving station after a message has been sent. If the response timer expires, the program on the next higher level normally retransmits the message frame (if the retransmit count has not yet expired). This program transmits continuously until the processor is reset or interrupted by an external source.

```
LOC OBJ CODE M STMT SOURCE STATEMENT
1 , SIO SDLC TEST PROGRAM
2
3 ,[0] 01-21-81/MDP INITIAL CREATION
4
```

**CONCLUSION**

This brief describes implementation of the SDLC protocol using the SIO in an interrupt-driven environment. Description for transmit and receive operations are given for use with simple control frame sequences. For frames that transfer data, the sequences are similar except for transmit, where a data character is sent to the SIO for a TBE interrupt. For receive, multiple RCA interrupts occur for each data byte received. The Z80 SIO enhances system performance by minimizing CPU intervention during data transfers using the SDLC protocol. Performance can be improved further by using the Z80 DMA with the SIO, resulting in an efficient system configuration that reduces CPU interaction to a minimum.
THIS PROGRAM SENDS ADDRESS 9EH, CONTROL 19H, AND DATA 81H CONTINUOUSLY USING THE ZBO VECTORED INTERRUPT MODE. THE SIO IS INITIALIZED TO USE SDLC WITH THE BAUD RATE CLOCK SUPPLIED BY HARDWARE INTERNAL TO THE SYSTEM.

EQUATES

ADDRESS: EQU 9EH ; ADDRESS FIELD
CTRL: EQU 19H ; CONTROL FIELD
DATA: EQU 81H ; INFORMATION FIELD
MSGLEN: EQU 1 ; MESSAGE LENGTH
RAM: EQU 2000H ; RAM ORIGIN
RAMSIZE: EQU 1000H ; RAM SIZE
SIODA: EQU 0 ; SIO PORT A DATA
SIOCA: EQU SIODA+1 ; SIO PORT A CTRL
SIODB: EQU SIODA+2 ; SIO PORT B DATA
SIOCB: EQU SIODA+3 ; SIO PORT B CTRL
CIOD: EQU 8 ; CI PORT C
CIOA: EQU CIOD+1 ; CI PORT B
CIOB: EQU CIOD+2 ; CI PORT A
CIOCTL: EQU CIOD+3 ; CI CTRL PORT
BAUD: EQU 9600 ; ASYNC BAUD RATE
RATE: EQU BAUD/100
CIODNT: EQU 9216/RATE
LITE: EQU 0EH ; LIGHT PORT
RSPCNT: EQU 100 ; RESPONSE TIMER VALUE

SIO PORT A DATA
CTRL
SIO PORT A

SIO PORT B DATA
CTRL
SIO PORT B

CI PORT C
CI PORT A
CI PORT B

ASYNC BAUD RATE

LIGHT PORT

CH. RESET CMD
ESC RESET CMD
TBE RESET CMD
RETI CH: A

ENAB. INT. NEXT RX
SRC RESET CMD
RX CRC RESET CMD
TX CRC RESET CMD

WAIT/READY
READY FUNCTION
WAIT/RDY ON RX
RX INT. FIRST CHAR
RX INT. ALL + PARITY
RX INT. ALL

STATUS AFFECTS VECT (CH. B ONLY)

TX INT. ENABLE
EXT INT. ENABLE

RX 8 BITS
RX 6 BITS
RX 7 BITS
RX 5 BITS

AUTO ENABLES
HUNT MODE
RX CRC ENABLE
ADDR SEARCH
ASYNC LOAD INHIBIT
RX ENABLE

64X CLOCK
32X CLOCK
16X CLOCK
1X CLOCK
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ CODE</th>
<th>M STMT</th>
<th>SOURCE STATEMENT</th>
<th>ASM 5 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>77</td>
<td></td>
<td>ENUM</td>
<td>EXTSYN: EQU 30H</td>
<td>EXT SYNC ENABLE</td>
</tr>
<tr>
<td>78</td>
<td></td>
<td>ENUM</td>
<td>SDLC: EQU 20H</td>
<td>SDLC MODE</td>
</tr>
<tr>
<td>79</td>
<td></td>
<td>ENUM</td>
<td>SYN6: EQU 1OH</td>
<td>16 BIT SYN</td>
</tr>
<tr>
<td>80</td>
<td></td>
<td>ENUM</td>
<td>SYNO: EQU 0</td>
<td>8 BIT SYN</td>
</tr>
<tr>
<td>81</td>
<td></td>
<td>ENUM</td>
<td>STOP2: EQU 0CH</td>
<td>2 STOP BITS</td>
</tr>
<tr>
<td>82</td>
<td></td>
<td>ENUM</td>
<td>STOP15: EQU 8</td>
<td>1.5 STOP BITS</td>
</tr>
<tr>
<td>83</td>
<td></td>
<td>ENUM</td>
<td>STOP1: EQU 4</td>
<td>1 STOP BIT</td>
</tr>
<tr>
<td>84</td>
<td></td>
<td>ENUM</td>
<td>SYNCEN: EQU 0</td>
<td>SYNC ENABLE</td>
</tr>
<tr>
<td>85</td>
<td></td>
<td>ENUM</td>
<td>EVEN: EQU 2</td>
<td>EVEN PARITY</td>
</tr>
<tr>
<td>86</td>
<td></td>
<td>ENUM</td>
<td>PARITY: EQU 1</td>
<td>PARITY ENABLE</td>
</tr>
<tr>
<td>87</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>88</td>
<td></td>
<td>ENUM</td>
<td>SIDWR5: EQU 5</td>
<td></td>
</tr>
<tr>
<td>89</td>
<td></td>
<td>ENUM</td>
<td>DTR: EQU 80H</td>
<td>ACTIVATE DTR</td>
</tr>
<tr>
<td>90</td>
<td></td>
<td>ENUM</td>
<td>TXB: EQU 6OH</td>
<td>TX 8 BITS</td>
</tr>
<tr>
<td>91</td>
<td></td>
<td>ENUM</td>
<td>TX6: EQU 4OH</td>
<td>TX 6 BITS</td>
</tr>
<tr>
<td>92</td>
<td></td>
<td>ENUM</td>
<td>TX7: EQU 2OH</td>
<td>TX 7 BITS</td>
</tr>
<tr>
<td>93</td>
<td></td>
<td>ENUM</td>
<td>TX5: EQU 0</td>
<td>TX 5 BITS</td>
</tr>
<tr>
<td>94</td>
<td></td>
<td>ENUM</td>
<td>BREAK: EQU 1OH</td>
<td>TX BREAK</td>
</tr>
<tr>
<td>95</td>
<td></td>
<td>ENUM</td>
<td>TXEN: EQU 8</td>
<td>TX ENABLE</td>
</tr>
<tr>
<td>96</td>
<td></td>
<td>ENUM</td>
<td>CRC16: EQU 4</td>
<td>CRC-16 MODE</td>
</tr>
<tr>
<td>97</td>
<td></td>
<td>ENUM</td>
<td>RTS: EQU 2</td>
<td>ACTIVATE RTS</td>
</tr>
<tr>
<td>98</td>
<td></td>
<td>ENUM</td>
<td>TXCRC: EQU 1</td>
<td>TX CRC ENABLE</td>
</tr>
<tr>
<td>99</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>ENUM</td>
<td>SIDWR6: EQU 6</td>
<td>LDW SYNC OR ADDR</td>
</tr>
<tr>
<td>101</td>
<td></td>
<td>ENUM</td>
<td>SIDWR7: EQU 7</td>
<td>HIGH SYNC OR FLAG</td>
</tr>
<tr>
<td>102</td>
<td></td>
<td>ENUM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>103</td>
<td></td>
<td>ENUM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>104</td>
<td></td>
<td>ENUM</td>
<td>SIOFLG = FLAGS FOR SIO STATUS</td>
<td></td>
</tr>
<tr>
<td>105</td>
<td></td>
<td>ENUM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>106</td>
<td></td>
<td>ENUM</td>
<td>BIT --- SET CONDITION</td>
<td></td>
</tr>
<tr>
<td>107</td>
<td></td>
<td>ENUM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>108</td>
<td></td>
<td>ENUM</td>
<td>0 TX ACTIVE</td>
<td></td>
</tr>
<tr>
<td>109</td>
<td></td>
<td>ENUM</td>
<td>1 MESSAGE COMPLETE</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
<td>ENUM</td>
<td>2 CTS ACTIVE</td>
<td></td>
</tr>
<tr>
<td>111</td>
<td></td>
<td>ENUM</td>
<td>3 DCD ACTIVE</td>
<td></td>
</tr>
<tr>
<td>112</td>
<td></td>
<td>ENUM</td>
<td>4 ABORT DETECT</td>
<td></td>
</tr>
<tr>
<td>113</td>
<td></td>
<td>ENUM</td>
<td>5 RX OVERRUN ERROR</td>
<td></td>
</tr>
<tr>
<td>114</td>
<td></td>
<td>ENUM</td>
<td>6 RX CRC ERROR</td>
<td></td>
</tr>
<tr>
<td>115</td>
<td></td>
<td>ENUM</td>
<td>7 RX END OF FRAME</td>
<td></td>
</tr>
<tr>
<td>116</td>
<td></td>
<td>ENUM</td>
<td>*E</td>
<td></td>
</tr>
<tr>
<td>117</td>
<td></td>
<td>ENUM</td>
<td>*** MAIN PROGRAM ***</td>
<td></td>
</tr>
<tr>
<td>119</td>
<td></td>
<td>ENUM</td>
<td>0000 ORG 0 ORG 0000 C32000 JP BEGIN JQ MAIN PROGRAM</td>
<td></td>
</tr>
<tr>
<td>121</td>
<td></td>
<td>ENUM</td>
<td>123 INTERRUPT VECTORS</td>
<td></td>
</tr>
<tr>
<td>122</td>
<td></td>
<td>ENUM</td>
<td>(MUST START ON EVEN BOUNDARY)</td>
<td></td>
</tr>
<tr>
<td>123</td>
<td></td>
<td>ENUM</td>
<td>0010 ORG 126 ORG * AND OFFFOH OR 10H</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td></td>
<td>ENUM</td>
<td>128 INTVEC:</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
<td>ENUM</td>
<td>SIOVEC:</td>
<td></td>
</tr>
<tr>
<td>129</td>
<td></td>
<td>ENUM</td>
<td>0010 9CO0 129 DEFW CHBTBE</td>
<td></td>
</tr>
<tr>
<td>130</td>
<td></td>
<td>ENUM</td>
<td>0012 D100 130 DEFW CHBESC</td>
<td></td>
</tr>
<tr>
<td>131</td>
<td></td>
<td>ENUM</td>
<td>0014 0101 131 DEFW CHBREQ</td>
<td></td>
</tr>
<tr>
<td>132</td>
<td></td>
<td>ENUM</td>
<td>0016 0F01 132 DEFW CHBSRC</td>
<td></td>
</tr>
<tr>
<td>133</td>
<td></td>
<td>ENUM</td>
<td>0018 3B01 133 DEFW CHATBE</td>
<td></td>
</tr>
<tr>
<td>134</td>
<td></td>
<td>ENUM</td>
<td>001A 4301 134 DEFW CHAESC</td>
<td></td>
</tr>
<tr>
<td>135</td>
<td></td>
<td>ENUM</td>
<td>001C 4801 135 DEFW CHARCA</td>
<td></td>
</tr>
<tr>
<td>136</td>
<td></td>
<td>ENUM</td>
<td>001E 5101 136 DEFW CHASRC</td>
<td></td>
</tr>
<tr>
<td>137</td>
<td></td>
<td>ENUM</td>
<td>138 BEGIN:</td>
<td></td>
</tr>
<tr>
<td>139</td>
<td></td>
<td>ENUM</td>
<td>0020 314020 139 LD SP,STAK ; INIT SP</td>
<td></td>
</tr>
<tr>
<td>140</td>
<td></td>
<td>ENUM</td>
<td>0023 ED5E 140 IM 2 ; VECTOR INTERRUPT MODE</td>
<td></td>
</tr>
<tr>
<td>141</td>
<td></td>
<td>ENUM</td>
<td>0025 3EOO 141 LD A,INTVEC/256 ; UPPER VECTOR BYTE</td>
<td></td>
</tr>
<tr>
<td>142</td>
<td></td>
<td>ENUM</td>
<td>0027 ED47 142 LD I,A</td>
<td></td>
</tr>
<tr>
<td>143</td>
<td></td>
<td>ENUM</td>
<td>0029 214520 143 LD HL,BUFFER</td>
<td></td>
</tr>
<tr>
<td>144</td>
<td></td>
<td>ENUM</td>
<td>002C 369E 144 LD (HL),ADDRESS ; STORE ADDRESS</td>
<td></td>
</tr>
<tr>
<td>145</td>
<td></td>
<td>ENUM</td>
<td>002E 23 145 INC HL</td>
<td></td>
</tr>
<tr>
<td>146</td>
<td></td>
<td>ENUM</td>
<td>002F 3619 146 LD (HL),CTRL ; STORE CTRL BYTE</td>
<td></td>
</tr>
<tr>
<td>147</td>
<td></td>
<td>ENUM</td>
<td>0031 23 147 INC HL</td>
<td></td>
</tr>
<tr>
<td>148</td>
<td></td>
<td>ENUM</td>
<td>0032 3681 148 LD (HL),DATA ; STORE DATA BYTE</td>
<td></td>
</tr>
<tr>
<td>149</td>
<td></td>
<td>ENUM</td>
<td>0034 CD4C00 149 CALL INIT ; INIT DEVICES</td>
<td></td>
</tr>
</tbody>
</table>
LOC OBJ CODE M STMT SOURCE STATEMENT ASM 5.9
0037 218720 150 LD HL, RBUF ; SETUP READ BUFFER
003A 228520 151 LD (RBPTR), HL
003D 213D00 152 LOOP: LD HL, LOOP ; SETUP STACK FOR RETURN
0040 E5 154 PUSH HL
0041 CD7D00 155 CALL WAKE ; WAKE TX
0044 3A4020 156 LD A, (SIOFLG) ; CHECK TX ACTIVE FLAG
0047 CB47 158 BIT 0, A
0049 20F9 159 JR NZ, LOOP1 ; LOOP IF TX ACTIVE
004B C9 160 RET
004C 217001 161 INIT: LD HL, SIOTA ; INIT CH A
004F 0E01 162 LD C, SIOCA
0051 060A 166 LD B, SIDEB-SIOTA
0053 EBD3 167 DTIR
0055 217A01 168 LD HL, SIOTB ; INIT CH B
0058 0E03 169 LD C, SIOCB
005A 0610 170 LD B, SIDEB-SIOTB
005C EBD3 171 DTIR
005E 3E00 172 LD A, 0 ; CLEAR FLAG BYTE
0060 324020 173 LD (SIOFLG), A
0063 DB0B 175 IN A, (CIOCTL) ; INSURE STATE 0
0065 AF 176 XOR A ; POINT TO REG 0
0066 D30B 177 OUT (CIOCTL), A ; CLEAR RESET OR STATE 0
0068 AF 179 XOR A
006A D30B 180 OUT (CIOCTL), A ; POINT TO REG 0
006C 3C 181 INC A ; WRITE RESET
006E D30B 182 OUT (CIOCTL), A
0070 AF 183 XOR A ; CLEAR RESET COND
0071 D30B 184 OUT (CIOCTL), A
0073 218A01 185 LD HL, CLST ; INIT CIO
0076 060E 186 LD B, CEND-CLST
0078 060B 187 LD C, CIOCTL
007A EBD3 188 DTIR
007C C9 189 RET
007D 3A4020 192 LD A, (SIOFLG) ; SET ACTIVE FLAG
0080 CBC7 193 SET 0, A
0082 324020 194 LD (SIOFLG), A
0085 214520 195 LD HL, BUFFER ; SET BUFFER PTR
0088 224320 196 LD (BUFPTR), HL
008B 3E03 197 LD A, 2+MSGLEN ; SET BYTE COUNT
008D 324120 198 LD (BYTES), A
0090 3E80 199 LD A, TCRCRE ; CLEAR TX CRC
0092 D303 200 OUT (SIOCB), A
0094 CD9C00 201 CALL CHTBSE ; START TRANSMIT
0097 3EC0 202 LD A, EOMRES ; RESET EOM LATCH
0099 D303 203 OUT (SIOCB), A
009B C9 204 RET
009C CD5901 210 CALL SAVE ; CH B TX BUFFER EMPTY
009F 214020 211 LD HL, SIOFLG ; POINT TO FLAG BYTE
00A2 CB4E 212 BIT 1, (HL) ; CHECK MC FLAG
00A4 201D 213 JR NZ, CHTB2 ; BRANCH IF MESSAGE COMPLET
00A6 3A4120 214 LD A, (BYTES) ; CHECK BYTE COUNT
00AB 215 215 OR A
00A9 280F 216 JR Z, CHTB1 ; BRANCH IF DATA DONE
00AC 3D 217 DEC A
00AD 324120 218 LD (BYTES), A
00B0 2A4320 219 LD HL, (BUFPTR)
00B3 7E 220 LD A, (HL)
00B4 D302 221 OUT (SIOCB), A

617-1564-0007 2-79 2-25-81
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ CODE</th>
<th>M STATE</th>
<th>SOURCE STATEMENT</th>
<th>ASM 5 9</th>
</tr>
</thead>
<tbody>
<tr>
<td>00B6</td>
<td>23</td>
<td>222</td>
<td>INC HL</td>
<td></td>
</tr>
<tr>
<td>00B7</td>
<td>224320</td>
<td>223</td>
<td>LD (BUF PTR), HL</td>
<td></td>
</tr>
<tr>
<td>00B8</td>
<td>C9</td>
<td>224</td>
<td>RET</td>
<td></td>
</tr>
<tr>
<td>00B9</td>
<td>CBE</td>
<td>226</td>
<td>SET 1, (HL)</td>
<td>CHBTB1:</td>
</tr>
<tr>
<td>00BD</td>
<td>3EC0</td>
<td>227</td>
<td>LD A, EOMRES</td>
<td></td>
</tr>
<tr>
<td>00BF</td>
<td>D303</td>
<td>228</td>
<td>OUT (SIOC B), A</td>
<td></td>
</tr>
<tr>
<td>00C1</td>
<td>1809</td>
<td>229</td>
<td>JR CHBTB3</td>
<td></td>
</tr>
<tr>
<td>00C3</td>
<td>CBE</td>
<td>231</td>
<td>RES 1, (HL)</td>
<td>CHBTB2:</td>
</tr>
<tr>
<td>00C5</td>
<td>CBE6</td>
<td>232</td>
<td>RES 0, (HL)</td>
<td></td>
</tr>
<tr>
<td>00C7</td>
<td>3E64</td>
<td>233</td>
<td>LD A, RSPCNT</td>
<td></td>
</tr>
<tr>
<td>00C9</td>
<td>324220</td>
<td>234</td>
<td>LD (SPTMR), A</td>
<td></td>
</tr>
<tr>
<td>00CC</td>
<td>3E28</td>
<td>236</td>
<td>LD A, TBERES</td>
<td></td>
</tr>
<tr>
<td>00CE</td>
<td>D303</td>
<td>237</td>
<td>OUT (SIOC B), A</td>
<td></td>
</tr>
<tr>
<td>00D0</td>
<td>C9</td>
<td>238</td>
<td>RET</td>
<td></td>
</tr>
<tr>
<td>00D1</td>
<td>CD5901</td>
<td>241</td>
<td>CALL SAVE</td>
<td></td>
</tr>
<tr>
<td>00D4</td>
<td>214020</td>
<td>242</td>
<td>LD HL, SIOFLG</td>
<td>CHBESC:</td>
</tr>
<tr>
<td>00D7</td>
<td>CB96</td>
<td>243</td>
<td>RES 2, (HL)</td>
<td></td>
</tr>
<tr>
<td>00D9</td>
<td>CB9E</td>
<td>244</td>
<td>RES 3, (HL)</td>
<td></td>
</tr>
<tr>
<td>00DB</td>
<td>CB65</td>
<td>245</td>
<td>RES 4, (HL)</td>
<td></td>
</tr>
<tr>
<td>00DF</td>
<td>D803</td>
<td>246</td>
<td>IN A, (SIOC B)</td>
<td></td>
</tr>
<tr>
<td>00E0</td>
<td>CB5B</td>
<td>248</td>
<td>BIT 3, B</td>
<td></td>
</tr>
<tr>
<td>00E2</td>
<td>C4FB00</td>
<td>249</td>
<td>CALL NZ, SETDCD</td>
<td></td>
</tr>
<tr>
<td>00E5</td>
<td>CB6B</td>
<td>250</td>
<td>BIT 5, B</td>
<td></td>
</tr>
<tr>
<td>00E7</td>
<td>C4FE00</td>
<td>251</td>
<td>CALL NZ, SETCTS</td>
<td></td>
</tr>
<tr>
<td>00EA</td>
<td>CB7B</td>
<td>252</td>
<td>BIT 7, B</td>
<td></td>
</tr>
<tr>
<td>00EC</td>
<td>C4FB00</td>
<td>253</td>
<td>CALL NZ, SETABT</td>
<td></td>
</tr>
<tr>
<td>00EF</td>
<td>CB4E</td>
<td>254</td>
<td>BIT 1, (HL)</td>
<td></td>
</tr>
<tr>
<td>00F1</td>
<td>2800</td>
<td>255</td>
<td>JR 2, CHBES1</td>
<td></td>
</tr>
<tr>
<td>00F3</td>
<td>3E10</td>
<td>257</td>
<td>LD A, ESCRES</td>
<td></td>
</tr>
<tr>
<td>00F5</td>
<td>D303</td>
<td>258</td>
<td>OUT (SIOC B), A</td>
<td></td>
</tr>
<tr>
<td>00F7</td>
<td>C9</td>
<td>259</td>
<td>RET</td>
<td></td>
</tr>
<tr>
<td>00F8</td>
<td>CBE6</td>
<td>261</td>
<td>SET 4, (HL)</td>
<td></td>
</tr>
<tr>
<td>00FA</td>
<td>C9</td>
<td>262</td>
<td>RET</td>
<td>SETABT:</td>
</tr>
<tr>
<td>00FB</td>
<td>CBDE</td>
<td>264</td>
<td>SET 3, (HL)</td>
<td></td>
</tr>
<tr>
<td>00FD</td>
<td>C9</td>
<td>265</td>
<td>RET</td>
<td>SETDCD:</td>
</tr>
<tr>
<td>00FE</td>
<td>CB6D</td>
<td>267</td>
<td>SET 2, (HL)</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>C9</td>
<td>268</td>
<td>RET</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>CD5901</td>
<td>271</td>
<td>CALL SAVE</td>
<td>CHBRC:</td>
</tr>
<tr>
<td>0104</td>
<td>D802</td>
<td>272</td>
<td>IN A, (SIOC B)</td>
<td></td>
</tr>
<tr>
<td>0106</td>
<td>2A8520</td>
<td>273</td>
<td>LD HL (RB PTR)</td>
<td></td>
</tr>
<tr>
<td>0109</td>
<td>77</td>
<td>274</td>
<td>LD (HL), A</td>
<td></td>
</tr>
<tr>
<td>010A</td>
<td>23</td>
<td>275</td>
<td>INC HL</td>
<td></td>
</tr>
<tr>
<td>010B</td>
<td>228520</td>
<td>276</td>
<td>LD (RB PTR), HL</td>
<td></td>
</tr>
<tr>
<td>010E</td>
<td>C9</td>
<td>277</td>
<td>RET</td>
<td></td>
</tr>
<tr>
<td>010F</td>
<td>CD5901</td>
<td>280</td>
<td>CALL SAVE</td>
<td>CHBSRC:</td>
</tr>
<tr>
<td>0112</td>
<td>3E01</td>
<td>281</td>
<td>LD A, I</td>
<td></td>
</tr>
<tr>
<td>0114</td>
<td>D303</td>
<td>282</td>
<td>OUT (SIOC B), A</td>
<td></td>
</tr>
<tr>
<td>0116</td>
<td>D803</td>
<td>283</td>
<td>IN A, (SIOC B)</td>
<td></td>
</tr>
<tr>
<td>0118</td>
<td>47</td>
<td>284</td>
<td>LD B, A</td>
<td></td>
</tr>
<tr>
<td>0119</td>
<td>214020</td>
<td>285</td>
<td>LD HL, SIOFLG</td>
<td></td>
</tr>
<tr>
<td>011C</td>
<td>CBB6</td>
<td>286</td>
<td>RES 6, (HL)</td>
<td></td>
</tr>
<tr>
<td>011E</td>
<td>CB7B</td>
<td>287</td>
<td>BIT 7, B</td>
<td></td>
</tr>
<tr>
<td>0120</td>
<td>C43B01</td>
<td>288</td>
<td>CALL NZ, SETEFF</td>
<td></td>
</tr>
<tr>
<td>0123</td>
<td>CB70</td>
<td>289</td>
<td>BIT 6, B</td>
<td></td>
</tr>
<tr>
<td>0125</td>
<td>C43501</td>
<td>290</td>
<td>CALL NZ, SETCRC</td>
<td></td>
</tr>
<tr>
<td>012B</td>
<td>CB6B</td>
<td>291</td>
<td>BIT 5, B</td>
<td></td>
</tr>
<tr>
<td>012A</td>
<td>C43201</td>
<td>292</td>
<td>CALL NZ, SETOVR</td>
<td></td>
</tr>
<tr>
<td>012D</td>
<td>3E30</td>
<td>294</td>
<td>LD A, SRCRES</td>
<td></td>
</tr>
</tbody>
</table>
LOC | OBJ CODE | M | STMT | TEST SDLC | ASM 5 9
---|---------|---|------|----------|---------
012F | D303 | 295 | OUT | (SIODA), A |
0131 | C9 | 296 | RET | |
0132 | CBEE | 298 | SET | 5, (HL) |
0134 | C9 | 299 | RET | |
0135 | CBF6 | 301 | SET | 6, (HL) |
0137 | C9 | 302 | RET | |
0138 | CBFE | 304 | SET | 7, (HL) |
013A | C9 | 305 | RET | |
013B | CD5901 | 308 | CALL | SAVE | ;CH, A TX BUFFER EMPTY |
013E | 3E28 | 309 | LD | A, TBERES |
0140 | D301 | 310 | OUT | (SIODA), A |
0142 | C9 | 311 | RET | |
0143 | CD5901 | 314 | CALL | SAVE | ;CH, A EXTERNAL/STATUS CHG |
0146 | 3E10 | 315 | LD | A, ESCRES |
0148 | D301 | 316 | OUT | (SIODA), A |
014A | C9 | 317 | RET | |
014B | CD5901 | 320 | CALL | SAVE | ;CH, A RX CHAR AVAIL |
014E | D300 | 321 | IN | A, (SIODA) |
0150 | C9 | 322 | RET | |
0151 | CD5901 | 324 | CALL | SAVE | ;CH, D SPECIAL RX Cond |
0154 | 3E30 | 326 | LD | A, SRCRES |
0156 | D301 | 327 | OUT | (SIODA), A |
0158 | C9 | 328 | RET | |
0159 | E3 | 333 | EX | (SP), HL | ;SP = HL |
015A | D5 | 334 | PUSH | DE | |
015B | C5 | 335 | PUSH | BC | |
015C | F5 | 336 | PUSH | AF | |
015D | DDE5 | 337 | PUSH | IX | |
015F | FDE5 | 338 | PUSH | IY | |
0161 | CD6F01 | 339 | CALL | GO | |
0164 | FDE1 | 340 | POP | IY | |
0166 | DDE1 | 341 | POP | IX | |
0168 | F1 | 342 | POP | AF | |
0169 | C1 | 343 | POP | BC | |
016A | D1 | 344 | POP | DE | |
016B | E1 | 345 | POP | HL | |
016C | FB | 346 | EI | |
016D | ED4D | 347 | RETI | |
016F | E9 | 349 | GO | |
0170 | 00 | 350 | JP | (HL) | |
0171 | 01 | 351 | *E | |
0172 | 01 | 352 | CONSTANTS | |
0173 | 02 | 353 | | |
0174 | 04 | 354 | SIOTA: | |
0176 | 05 | 355 | | |
0177 | AA | 356 | DEFB | SIOWRO | ;CHAN RESET |
0178 | 03 | 357 | DEFB | CHRES | |
0179 | 01 | 358 | DEFB | SIOWRI | ;CHAN CHARACS |
017A | 02 | 359 | DEFB | WREN+RDI+RXIAI+TXI | |
017C | 04 | 360 | DEFB | SIOWR4 | ;MODE |
017E | 05 | 361 | DEFB | X16+STOP2+EVEN+PARITY | |
017F | 07 | 362 | DEFB | SIOWR5 | ;TX PARAMS. |
0183 | 03 | 363 | DEFB | DTR+TX7+TXEN+RTS | |
0185 | 04 | 364 | DEFB | SIOWR3 | ;RX PARAMS. |
0187 | 01 | 365 | DEFB | RX7+RXEN | |
0189 | 41 | 366 | SIODA: | EQU | * |
      | 367 | | | |
### TEST SDLC

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ CODE M STMT</th>
<th>SOURCE STATEMENT</th>
<th>ASM 5</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>017A 00 369</td>
<td>DEFB SIOWR0</td>
<td>;CHAN RESET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>017B 18 370</td>
<td>DEFB CHRES</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>017C 02 371</td>
<td>DEFB SIOWR2</td>
<td>;VECTOR REG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>017D 10 372</td>
<td>DEFB SIOVEC. AND. 255</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>017E 04 373</td>
<td>DEFB SIOWR4</td>
<td>;MODE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>017F 20 374</td>
<td>DEFB X1+SDLC+SYNCEN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0180 01 375</td>
<td>DEFB SIOWR1</td>
<td>;CHAN CHARACS.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0181 1F 376</td>
<td>DEFB RXIA+SIOSAV+TXI+EXTI</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0182 06 377</td>
<td>DEFB SIOWR6</td>
<td>;ADDRESS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0183 9E 378</td>
<td>DEFB ADDRESS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0184 07 379</td>
<td>DEFB SIOWR7</td>
<td>;FLAG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0185 7E 380</td>
<td>DEFB 01111110B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0186 05 381</td>
<td>DEFB SIOWR5</td>
<td>;TX PARAMS.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0187 EB 382</td>
<td>DEFB DTR+TXB+TXEN+RTS+TXCRC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0188 03 383</td>
<td>DEFB SIOWR3</td>
<td>;RX PARAMS.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0189 C1 384</td>
<td>DEFB RXB+RXEN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>385</td>
<td>SIOEB: EQU $</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>386</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>387</td>
<td>CLST:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>018A 2B 388</td>
<td>DEFB 2BH</td>
<td>;PORT B MODE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>018B 00 389</td>
<td>DEFB 00000000B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>018C 2B 390</td>
<td>DEFB 2BH</td>
<td>;DATA DIRECTION</td>
<td></td>
<td></td>
</tr>
<tr>
<td>018D EE 391</td>
<td>DEFB 11101110B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>018E 1C 392</td>
<td>DEFB 1CH</td>
<td>;CT1 MODE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>018F C2 393</td>
<td>DEFB 11000010B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0190 16 394</td>
<td>DEFB 16H</td>
<td>;CT1 TC MSB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0191 00 395</td>
<td>DEFB 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0192 17 396</td>
<td>DEFB 17H</td>
<td>;LSB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0193 60 397</td>
<td>DEFB CI0CNT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0194 01 398</td>
<td>DEFB 1</td>
<td>;MASTER CONFIG. REG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0195 F0 399</td>
<td>DEFB 11110000B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0196 0A 400</td>
<td>DEFB 10</td>
<td>;CT1 TRIGGER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0197 06 401</td>
<td>DEFB 00000110B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>402</td>
<td>CEND: EQU $</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>403</td>
<td>*E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>404</td>
<td>, ; DATA AREA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>405</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>406</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>ORG RAM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>DEFS 64</td>
<td>;STACK AREA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2040</td>
<td>SIGAO: DEFS 1</td>
<td>;SIG FLAG BYTE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2041</td>
<td>BYTES: DEFS 1</td>
<td>;BUFFER BYTE COUNT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2042</td>
<td>RSPTMR: DEFS 1</td>
<td>;RESPONSE TIMER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2043</td>
<td>DEFS 2</td>
<td>;BUFFER POINTER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2045</td>
<td>BUFFER: DEFS 64</td>
<td>;BUFFER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2085</td>
<td>RBPTR: DEFS 2</td>
<td>;READ BUFF PTR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>416</td>
<td>RBUF: EQU $</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>417</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>418</td>
<td>END</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A popular communication protocol used to exchange information between data processing devices has been in use for some time. This protocol, developed by IBM, is called binary synchronous protocol, or bisync. The Z80 SIO provides a flexible and powerful tool for the implementation of the bisync protocol. However, there are some design considerations that require special attention. This paper will discuss these design considerations and offer an approach to using bisync with the Z80 SIO. Specific examples are presented and readers who are unfamiliar with the bisync protocol should refer to the ANSI standard (1) or the IBM publication (2) listed at the end of this paper.

Bisync is a character-oriented protocol with information transmitted in blocks between two (or more) data communication devices. The medium through which this information is conveyed is called the data link. The particular data link discussed in this paper is a point-to-point link using the ASCII transmission code. Other codes, such as EBCDIC, are not covered, but the format for bisync is basically the same. The data link consists of a master station (usually a computer) and a slave station (usually a terminal) with the associated communication gear in between—modems, phone lines, etc. The master station controls message flow by polling and selecting the slave station. Polling involves sending a general request message to the slave station(s) to determine whether or not any of the slaves have data to send (traffic). If a slave station has traffic, it responds to the poll and the master can then select that particular slave for information exchange. Slaves can only respond to a master device and cannot initiate communication on the data link.

Information is exchanged by means of a well-defined block structure. Message blocks consist of a header, body, and trailer (Figure 1). The header is made of two or more SYN characters (hence the name bisync), a start of header (SOH) character, and addressing and control information for a particular slave station.

The body begins with a start of text (STX) character and encompasses the entire text information. The body generally contains ASCII text data, although 8-bit binary data can be transmitted using transparent text mode.

The trailer contains the end of text (ETX) character and the block check character (BCC). The BCC is used for detecting errors through "cyclic redundancy checking" (CRC) or "longitudinal redundancy checking" (LRC).

Error detection is essential when transferring information between data processing equipment. Since ASCII specifies only seven bits for its code, the eighth bit is used for vertical redundancy checking (VRC), more commonly known as character parity. In synchronous communications, character parity is generally odd, whereas in asynchronous communications it is even. Figure 2 shows typical ASCII characters with parity. The SIO can be programmed for 7-bit characters with odd parity enabled to minimize software overhead.
Because VRC applies only to the individual character, the entire message block has an LRC that makes up the BCC. The LRC is a simple bit position checksum where the number of 1s for each position (0 through 6) is even for a block of data. Since the BCC is a character, LRC is subject to the same character parity rules as the rest of the data block. The LRC includes all characters, except SYN, starting with the first character after SOH or STX and up to and including ETX in the trailer (Figure 3). Since the SIO cannot calculate the LRC, the task is left up to the user. LRC can be generated on a microprocessor with little effort by taking the message block and XORing the data with an alternate character, the entire message block has an LRC. The remainder of this paper illustrates how to use the message block and XORing the data with an alternative character, the entire message block has an LRC.

Because VRC applies only to the individual character, the entire message block has an LRC that makes up the BCC. The LRC is a simple bit position checksum where the number of 1s for each position (0 through 6) is even for a block of data. Since the BCC is a character, LRC is subject to the same character parity rules as the rest of the data block. The LRC includes all characters, except SYN, starting with the first character after SOH or STX and up to and including ETX in the trailer (Figure 3). Since the SIO cannot calculate the LRC, the task is left up to the user. LRC can be generated on a microprocessor with little effort by taking the message block and XORing the data with an alternate character, the entire message block has an LRC.

Another type of BBC is generated by a cyclic redundancy check (CRC), which results in a more powerful method of block checking. CRC-12 is used for 6-bit transmission code and CRC-16 is used for 8-bit transmission code. CRC is used in lieu of character parity and LRC, as with transparent text mode operation.

The remainder of this paper illustrates how to use the SIO in three special cases of the bisync protocol: transparent text mode, abort/interrupt procedures, and error recovery procedures.

Transparent text mode is useful in bisync when information exchanged between master and slave is not ASCII data. For example, a binary data file (object program) might be sent from master to slave. ASCII transmission code is only seven bits long making it difficult to send 8-bit binary data. One alternative is to convert the binary data to ASCII hex format at the master, transmit it to the slave and reconvert it back into binary at the slave. However, two disadvantages result from this. First, the master and slave require a means of conversion, by either software or hardware, adding cost to the data link. Since the slave (terminal) is burdened most by this, such an approach is usually not feasible. The other disadvantage is that the exchange of information is slower since two (or more) ASCII characters are sent for every eight bits of binary data. The bisync protocol has provisions for sending 8-bit binary data by using transparent text mode transmission. In this mode, character parity is disabled, allowing the full eight bits to be used for data. However, to allow control within the constraints of the protocol, there are certain limitations on the binary data pattern. The primary difference is that during transparent mode some communication control characters are preceded by a DLE character, actually making the control characters a two-character sequence. To distinguish a data byte from a control DLE, the protocol specifies insertion of another DLE. The receiver then throws away the first DLE, keeping the second as data. Table 1 shows the communication control characters that are valid during transparent mode.

Another character change occurs when the SYN character is used for line fill. Normally, the SYN character is ignored, but during transparent mode the SYN is preceded by a DLE, and both are consequently ignored by the receiver. In the event that the CPU does not have a character ready to send, the SIO automatically inserts SYN characters into the data stream. With the SIO programmed for 16-bit sync characters, two syncs are sent from the SIO (write registers WR6 and WR7) when its transmit buffer is empty. In transparent mode, the user must change WR6 and WR7 to DLE, SYN in order for the SIO to provide the proper line fill characters. In accordance with the ANSI standard, line fill characters are not included in the SIO CRC calculation during transmit. During reception in transparent mode, the software must disable CRC accumulation when the DLE SYN character sequence is detected.

While in transparent mode, the user must be concerned with the error detection codes. If parity is enabled in the SIO normally, it must be disabled during transparent mode. This change in SIO operation affects both transmit and receive and should therefore be considered if using full duplex.

| Table 1. Control Codes Used In Transparent Mode |
|-------------------------------|----------------|
| DLE  | STX                 | Start of transparent text |
| DLE  | ETB                 | End of transparent text block |
| DLE  | ETX                 | End of transparent text |
| DLE  | SYN                 | Idle sync |
| DLE  | ENQ                 | Enquiry |
| DLE  | DLE                 | DLE data |
| DLE  | SOH                 | Start of transparent header |
Since the S10 allows CRC enable/disable on the fly, the software can easily control CRC accumulation in both receive and transmit. During transmit, the CRC must be enabled/disabled before the character is transferred into the serial shift register. During receive, the CRC accumulation is delayed eight bits. After the character is transferred from the serial shift register into the buffer, the user has to read that character, decide whether or not to continue CRC accumulation, and disable/enable CRC before the next character is transferred to the buffer. This is not a generally a problem, since character transfers occur about every 833 microseconds at 9600 baud. Table 2 shows the characters included and omitted in the CRC during transparent mode.

<table>
<thead>
<tr>
<th>Omitted from CRC</th>
<th>Included in CRC</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLE</td>
<td>DLE of DLE DLE</td>
</tr>
<tr>
<td>DLE</td>
<td>ETX of DLE ETX</td>
</tr>
<tr>
<td>DLE</td>
<td>ETB of DLE ETB</td>
</tr>
<tr>
<td>DLE STX*</td>
<td>STX of DLE STX**</td>
</tr>
</tbody>
</table>

*If not preceded by transparent header  **If preceded by DLE within same block block

When CRC accumulation is to be resumed, the software should enable CRC before the desired character is transferred to the receive buffer. For example, suppose a DLE pair is received during transparent text mode. The S10 generates an interrupt when the first DLE is transferred to the receive buffer. The driver program reads the DLE and immediately disables CRC. When the next interrupt occurs, the driver reads the second DLE and immediately enables CRC to include the second DLE into the CRC accumulation.

The second category of interest includes abort and interrupt procedures. There are two types of aborts: block abort and sending station abort. There are three types of interrupts: termination interrupt, reverse interrupt and temporary interrupt.

The block abort is used by the sending station when, in the process of transmitting a data block, the sending station detects an error condition in the data and decides to terminate the block so that the receiving station will discard it. In nontransparent mode, block abort is accomplished by ending the block with an ENQ character, instead of ETX or ETB. The sending station then waits for a reply from the receiver, which should be a NAK. The transparent mode procedure is identical except that a DLE ENQ character sequence is used. Since a block abort puts the data back in nontransparent mode, NAK is the valid response the receiver should send in both transparent and nontransparent modes.

The sending station abort is similar to the block abort, except that the sending station does not necessarily do a block abort but simply sends the current message block, waits for a response or timeout, and then sends an EOT to regain control of the data link. The sending station abort is useful when transmission to a particular receiver is necessary due to a higher priority message, buffer overflow condition, error detection, etc. Once the sending station abort sequence is made, the master can perform any data link control function.

From the receiver side, a termination interrupt causes the sending station to stop transmission. Such a procedure is useful when the receiver cannot accept any more data or incurs an error condition, such as paper jam, card jam, hardware error, etc. To accomplish a termination interrupt, the receiving station sends an EOT instead of the normal response. The EOT resets all stations on the link and allows the master to issue any control sequence.

The reverse interrupt (RINT) is used when the receiving station needs to transmit during reception of several message blocks. The RINT occurs when a receiver detects a valid CRC or LRC and, instead of returning an ACK, sends a DLE "\" character sequence to signal an affirmative acknowledgement and to stop transmission of data. Some exceptions and a more detailed description of RINT can be found in the ANSI standard.

The temporary interrupt procedure, WACK (Wait Before Sending Positive Acknowledge), is used by the receiving station to indicate positive acknowledgement and an inability to receive more data. Such a response may be necessary when the receiving station cannot accept data continuously, such as during a printing operation. The WACK consists of a DLE "\" character sequence and is sent in place of an ACK or ACKn. The sending station then sends ENQs (Enquiry) until the receiving station stops sending WACKs. The sending station can resume transmitting data when the receiving station sends an ACK or ACKn.

Recovery procedures provide a means of preventing data link instability. The recovery mechanism consists mainly of timers, grouped into four basic areas, and a NAK counter. The NAK counter is used to prevent repeated NAKs from inhibiting further communications. The sending unit counts how many NAKs it receives for a particular data block so that after a predetermined number of retries, it can recover and pursue another course of
action. The particular count value and course of action taken when the count expires are left up to the user.

Four timers (timer A or response timer, timer B or receiver timer, timer C or gross timer, and timer D or no activity timer) prevent the data link from getting "hung" or going idle for extended periods of time. Generally, the shortest interval is used with timer A, and the longest interval is used with timer D. For maximum system efficiency, however, the receiver timer (timer B) should timeout before the response timer (timer A). The particular implementation of these timers varies from system to system, and some flexibility of exact timer values is left up to the user.

Since it is assumed that interrupts will be used with the S10, an interrupt driven receiver timer count is kept in memory and is reinitialized each time a character is received (receive interrupt). The same applies for the response timer, except that when a timeout occurs, the transmit driver has several options to follow.

If the S10 is set to transmit CRC on transmit underrun, then the driver could simply set its flags and not fill the buffer. This allows a normal exit, since the S10 will then send its CRC bytes. If the S10 is set to not transmit CRC on transmit underrun, then it sends sync characters (SYN SYN or DLE SYN, whichever was last written to WR6 and WR7) until the transmit buffer is filled or transmit data is set to marking.

In any event, enough time must be allowed after CRC is sent so that the receiver can properly decode CRC. Because of the character delay in the S10 during CRC accumulation, about 20 clock cycles are necessary after the last CRC byte is sent to ensure adequate decoding time. (See the S10 Technical Manual for further details.) The S10 could be programmed to send pad characters, either by disabling parity and sending 8-bit FFs (hex) or by filling WR6 and WR7 with FF hex. If enabled, the S10 automatically sends whatever is in its sync registers upon transmit underrun. Multiple message blocks do not have to be separated by pad characters as long as CRC is valid for the previous message block. However, to insure adequate time for the receiver to process CRC, it is recommended that at least two pad characters follow the last character of a block.

Using the S10 for the bisync protocol is fairly straightforward. Care should be exercised when using the S10 in transparent text mode, but the implementation is greatly simplified by the S10's flexibility, as compared to other serial communications ICs. The CRC capabilities of the S10 provide a powerful means of maintaining maximum data integrity with minimum software overhead. Coupled with the DMA and the interrupt capabilities of the Z80 processor, the user will find the S10 an excellent choice in serving data communication needs.

INTRODUCTION

Serial data communication is among the most widely used forms of exchanging information with and between computers. The rapid expansion of this form of communication has created the need for low-cost, efficient, and flexible peripheral devices that provide the user with a wide variety of options. The Z80 DART is designed to fill this need by providing two independently programmable, asynchronous communication channels for a Z80-based system.

This application brief describes the use of the Z80 DART in a Z80-based system. Further information on the Z80 CPU and Z80 DART is available in the Zilog Data Book (document number 00-2034-A), Z8400 Z80 CPU Product Specification (document number 00-2001-A), and the Z8470 Z80 DART Product Specification (document number 00-2044-A).

HARDWARE

The hardware for this application consists of a Z8400 Z80 CPU, Z8470 Z80A DART, Z8536 C10, 4K ROM, and 4K RAM. Figure 1 shows a block diagram of the system. The C10 supplies the bit rate clock for the DART and allows the baud rate for each channel to be determined by the software.

The DART-to-CPU interface consists of eight bidirectional data lines, seven control lines, and three daisy chain interrupt control lines. The data lines are used to transfer data between the DART and the CPU. The direction of data flow on the data lines is determined through the use of the CE, RD,
and IORQ control lines. When CE and IORQ are active, a data transfer occurs between the CPU and DART. If RD is active at the same time, data is sent from the DART to the CPU. If RD is not active, data is sent from the CPU to the DART. Ml signals an interrupt acknowledge cycle from the CPU in conjunction with IORQ. The RESET line performs a device reset on the DART, allowing it to be placed in a known state. The remaining two control lines determine which of the four ports are being accessed. Table 1 shows the relationship of these two lines to the ports.

Table 1. DART Port Addressing

<table>
<thead>
<tr>
<th>Port</th>
<th>C/D</th>
<th>B/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel A Data</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Channel B Data</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Channel A Control</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Channel B Control</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

C/D and B/A are usually tied to the lowest two CPU address lines used for I/O device selection. Figure 2 shows the device-select decode logic used in this application.

![Figure 2. DART Device Select Logic](image)

External connections to the Z80 DART include serial data and control lines and modem control lines. The serial data lines are Transmit Data (TxD) and Receive Data (RxD) for each channel. Separate transmit and receive clock inputs are available on channel A (TxCA and RxCA), while a combined transmit/receive clock input is provided for channel B (TxRXCB). To allow separate baud rates for both channels, TxCA and RxCA are tied together and connected to one counter/timer output, and TxRXCB is connected to another counter/timer output. This provides the user with a simple, software-programmable baud rate generator.

The modem control lines provide the user with a means of controlling some external device such as a modem. This is particularly useful for remote applications in which the CPU must determine a course of action based on the status of the modem control lines. For example, Ring Indicator (RI) can be used to signal the CPU that an incoming call needs to be answered, or Data Terminal Ready (DTR) can be used in conjunction with Data Carrier Detect (DCD) to signal the modem that data communications can take place. DTR remains active as long as the DART is communicating over the serial data link. The CPU can "hang up" or disconnect the telephone connection by deactivating DTR. Finally, Request To Send (RTS) and Clear To Send (CTS) are useful in a multidrop configuration; that is, when three or more modems are connected to the same telephone line RTS is used to switch the carrier for a particular modem on or off under software control. CTS is monitored so that after RTS is activated the CPU knows when to start sending data.

The IEl, IEO, and INT lines form the Z80 daisy-chain interrupt controls that enable proper interrupt sequencing. INT is an open-drain, active Low output that is connected to the Z80 CPU INT Input, along with a pullup resistor. IEl is usually connected to the preceding device in the daisy chain or is tied High if there is no preceding device. IEO is connected to the following device in the daisy chain or is left open. This application example uses interrupts with the Status Affects Vector (SAV) programming option. Interrupts are prioritized internally in the DART according to the various conditions. There are four separate interrupt groups for each channel. Table 2 shows the relative priorities of these interrupts.

Table 2. DART Interrupt Priority

<table>
<thead>
<tr>
<th>Priority</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highest</td>
<td>Ch. A Special Rx Condition</td>
</tr>
<tr>
<td></td>
<td>Ch. A Rx Char. Available</td>
</tr>
<tr>
<td></td>
<td>Ch. A Tx Buffer Empty</td>
</tr>
<tr>
<td></td>
<td>Ch. A External/Status Change</td>
</tr>
<tr>
<td></td>
<td>Ch. B Special Rx Condition</td>
</tr>
<tr>
<td></td>
<td>Ch. B Rx Char. Available</td>
</tr>
<tr>
<td></td>
<td>Ch. B Tx Buffer Empty</td>
</tr>
<tr>
<td>Lowest</td>
<td>Ch. B External/Status Change</td>
</tr>
</tbody>
</table>
### Programming

Programming the Z80 DART consists of two parts: initialization and program operation. Initialization includes defining the operating characteristics of the DART. This is done by writing a series of bytes to the control port of each channel. A detailed description of the programming for the DART can be found in the DART Product Specification (document number 00-2044-A). A listing containing an initialization routine for the DART can be found in the appendix of this brief.

Once initialized, the DART interrupts the CPU for certain conditions that occur. These conditions include Transmit Buffer Empty, Receive Character Available, Special Receive Condition, and External/Status Change for each channel.

The DART generates a Transmit Buffer Empty (TBE) interrupt when a character is transferred from the internal buffer to the shift register. The interrupt service routine determines whether to send another character to the DART or to issue a Reset Tx Interrupt Pending command. If a character is loaded into the DART, the interrupt condition is automatically removed. If a character is not loaded, the software issues a Reset Tx Interrupt Pending command to remove the interrupt condition and also sets an internal program status flag that signals the transmit channel as inactive. When transmission starts from an inactive condition (such as after initialization), the main program must activate the transmitter by sending a character to the DART. In this application, a call to the transmit interrupt service routine activates the transmitter after the buffer and pointers have been initialized.

The Receive Character Available (RCA) interrupt occurs after the DART transfers a character from the serial shift register to the receiver FIFO. The DART can store up to three characters in the FIFO, giving the CPU some flexibility in receive interrupt timing. Read Register 0 (RRO, bit 0) can be checked to see if any more characters are in the FIFO before exiting the interrupt service routine. If the DART is programmed so that parity does not affect the interrupt vector, parity errors must be checked in the receive service routine. This is done by writing a register pointer to the DART for Read Register 1 (RR1) and then reading the contents. The bit test instructions of the Z80 CPU are particularly useful in determining which bits are set or cleared. Processing for these errors is the same as for any receive condition.

The DART generates a Special Receive Condition (SRC) interrupt if it detects a parity error, overrun, or framing error during reception. When this occurs, the programmer should reset the error condition by issuing an Error Reset command to the DART. After the Error Reset command is issued, the programmer should read and discard the data if necessary. If the data is not discarded, then an RCA interrupt occurs immediately after exiting the SRC service routine.

An External/Status Change (ESC) interrupt occurs when the DART detects a change in the external signals (RI, CTS, DCD) or when a receive break condition is initiated or terminated. This is useful in monitoring the interface to the modem where a software flag is set when the break condition is detected and reset when the break condition is cleared. With CTS, DCD, and RI, the same procedure is followed as with a break condition. However, if the auto enable bit is set in the DART, the DART does not transmit data until CTS becomes active, nor does it receive data until DCD becomes active.

The appendix contains the listing of a test program for the DART. While it is by no means complete, it does highlight the interrupt features of the Z80 DART.

### Conclusion

As do other Z80 peripheral products, the Z80 DART interfaces well with the Z80 CPU. The software required to utilize the features of the DART is conducive to efficient programming. Interrupts provide a key method of maintaining efficient system operation, keeping CPU processing overhead to a minimum.

Other methods of utilizing the DART include a "polled" (noninterrupt) system. Because the Z80 CPU has three interrupt modes, the DART can be used with the CPU without vectored interrupts. However, such simplicity is usually at the expense of program size and speed.

Nevertheless, the user will find the Z80 DART a viable alternative to more expensive devices when considering the asynchronous communication requirements for any Z80 system.

### Appendix

Following is the listing of a DART test program. Note that all interrupt service routines are dummy routines, except DATBE, which transfers characters from the buffer to Port A transmitter.
NOTE. DARCA, DAESC, AND DASRC are dummy routines.

Figure 3. Flow Diagram for DART Test Program
DART TEST PROGRAM

EQUATES

RAM EQU 2000H
RAMSZ: EQU 1000H
CIOA EQU B
CIOB EQU CIOA+1
CIOC EQU CIOA+2
CIOCTL. EQU CIOA+3
BAUD EQU 9600
RATE EQU BAUD/100
CICNT EQU 576/RATE
DRTDA: EQU 4
DRTCA EQU DRTDA+1
DRTDB EQU DRTDA+2
DRTC B EQU DRTDA+3

DART PARAMETERS

DRTWRO: EQU 0
CHRES EQU 18H
ESCRE S EQU 10H
TBERES: EQU 28H
SRCRES EQU 30H
RETI A: EQU 3BH
ENINRX EQU 20H
DRTWR1. EQU 1
WREN EQU 8OH
RDY. EQU 40H
WRONR: EQU 20H
RXIFC: EQU 8
RXIA: EQU 18H
DRTSAV: EQU 4
TXI. EQU 2
EXTI EQU 1
DRTWR2. EQU 2
DRTWR3: EQU 3
RX8: EQU 0COH
RX6: EQU 80H
RX7: EQU 40H
RX5: EQU 0
AUTOEN: EQU 20H
RXEN EQU 1
DRTWR4: EQU 4
X64: EQU 0COH
X32: EQU 80H
X16: EQU 40H
X1: EQU 0
STOP2: EQU 0CH
STOP15 EQU 8
STOP1 EQU 4
EVEN EQU 2
PARITY EQU 1
DRTWR5 EQU 5
DIR EQU 8OH
TXB EQU 60H
TX6 EQU 40H
TX7: EQU 20H
TX5: EQU 0
BREAK EQU 10H
TXEN EQU 5
RTS EQU 2

#E
TEST. DART

LOC OBJ CODE M STM source STATEMENT ASM 5.9

73 , . *** MAIN PROGRAM *** 74

0000 075 ORG 0
0000 C3200 76 JP BEGIN ; GO MAIN PROGRAM
77
78 , INTERRUPT VECTORS 79

0010 80 ORG $, AND. OFFFOH OR. 10H
81 INTVEC:
82 DRTVEC:

0010 7E00 83 DEFW DBTBE
0012 9000 84 DEFW DBESC
0014 BA00 85 DEFW DBRCA
0016 A400 86 DEFW DBSRC
0018 B800 87 DEFW DATBE
001A D100 88 DEFW DAESE
001C B800 89 DEFW DARCA
001E E500 90 DEFW DASRC

BEGIN

0020 31B320 93 LD SP, STAK ; INIT SP.
0023 ED5E 94 IM 2 ; VECTOR INTERRUPT MODE
0025 3E00 95 LD A, INTVEC/256 ; UPPER VECTOR BYTE
0027 ED47 96 LD I, A
0029 CD4600 97 CALL INIT ; INIT DEVICES
002C 210020 98 LD HL, BUFFER
002F 063E 99 LD B, 62

BEGIN

0031 7B 101 LD A, B
0032 F640 102 OR 40H
0034 77 103 LD (HL), A
0035 23 104 INC HL
0036 10F9 105 DJNZ LOOP

003B 360D 106 LD (HL), 13 ; CR
0039 23 107 INC HL

003B 360A 108 LD (HL), 10 ; LF
003D 21020 109 LD HL, BUFFER
0040 22412 110 LD (BUFPR), HL

0043 CDB600 111 CALL DATBE ; WAKE TX

0046 18FE 113 JR $ ; LOOP FOREVER

0048 211001 115 INIT
004B 0E05 116 DRTINI:

004B 0E05 118 LD C, DRTCA
004D 060A 119 LD B, DRTEA-DRTTA
004F EDB3 120 QTIR

0051 211A01 121 LD HL, DRTTB ; INIT CH B

0054 0E07 122 LD C, DRTCB
0056 060C 123 LD B, DRTEB-DRTTB
0058 EDB3 124 QTIR

005A AF 125 XOR A ; CLEAR FLAG BYTE
005B 324020 126 LD (DRTFLG), A

005E D80B 128 IN A, (CIOCTL) ; INSURE STATE 0
0060 AF 129 XOR A  ; POINT TO REG 0

0061 D30B 130 OUT (CIOCTL), A
0063 D80B 131 IN A, (CIOCTL)
0065 AF 132 XOR A

0066 D30B 133 OUT (CIOCTL), A
0068 3C 134 INC A ; WRITE RESET

0069 D30B 135 OUT (CIOCTL), A
006B AF 136 XOR A ; ELSE, CLEAR RESET COND

006C D30B 137 OUT (CIOCTL), A
006E 3FE 138 LD A, OFEH ; (FUDGE FOR CIO QUIRK)

0070 D30B 139 OUT (CIOCTL), A
0072 D30B 140 OUT (CIOCTL), A

0074 212601 141 LD HL, CLST ; INIT CID

0077 0620 142 LD B, CEND-CLST
0079 0E0B 143 LD C, CIOCTL
TEST.DART

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ CODE</th>
<th>OBJ CODE</th>
<th>STMT</th>
<th>SOURCE</th>
<th>STATEMENT</th>
<th>ASM 5.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>007B</td>
<td>EDB3</td>
<td>144</td>
<td>OTIR</td>
<td>RET</td>
<td></td>
<td></td>
</tr>
<tr>
<td>007D</td>
<td>C9</td>
<td>145</td>
<td>RET</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>007E</td>
<td>CDF900</td>
<td>162</td>
<td>CALL</td>
<td>SAVE</td>
<td>,CH. B TX BUFFER EMPTY</td>
<td></td>
</tr>
<tr>
<td>0081</td>
<td>3E00</td>
<td>163</td>
<td>LD</td>
<td>A, DRTWO</td>
<td>POINT TO REG. 0</td>
<td></td>
</tr>
<tr>
<td>0083</td>
<td>D307</td>
<td>164</td>
<td>OUT</td>
<td>(DRTCB), A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0085</td>
<td>3E28</td>
<td>165</td>
<td>LD</td>
<td>A, TBERES</td>
<td>RESET TBE</td>
<td></td>
</tr>
<tr>
<td>0087</td>
<td>D307</td>
<td>166</td>
<td>OUT</td>
<td>(DRTCB), A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0089</td>
<td>C9</td>
<td>167</td>
<td>RET</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0090</td>
<td>CDF900</td>
<td>170</td>
<td>CALL</td>
<td>SAVE</td>
<td>,CH. B RX CHAR Avail.</td>
<td></td>
</tr>
<tr>
<td>0092</td>
<td>D306</td>
<td>171</td>
<td>IN</td>
<td>A, (DRTDB) ; READ DATA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0094</td>
<td>C9</td>
<td>172</td>
<td>RET</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0095</td>
<td>D307</td>
<td>173</td>
<td>CALL</td>
<td>SAVE</td>
<td>,CH. B EXTERNAL/STATUS</td>
<td></td>
</tr>
<tr>
<td>0097</td>
<td>3E10</td>
<td>174</td>
<td>LD</td>
<td>A, DRTWO</td>
<td>POINT TO REG. 0</td>
<td></td>
</tr>
<tr>
<td>0099</td>
<td>D307</td>
<td>175</td>
<td>OUT</td>
<td>(DRTCB), A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>009B</td>
<td>3A4020</td>
<td>176</td>
<td>LD</td>
<td>A, (DRTFLQ) ; UPDATE FLAG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>009D</td>
<td>CBE7</td>
<td>177</td>
<td>SET</td>
<td>4, A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00A0</td>
<td>324020</td>
<td>178</td>
<td>LD</td>
<td>(DRTFLG), A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00A2</td>
<td>C9</td>
<td>179</td>
<td>RET</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00A4</td>
<td>CDF900</td>
<td>180</td>
<td>CALL</td>
<td>SAVE</td>
<td>,CH. B SPECIAL RX COND.</td>
<td></td>
</tr>
<tr>
<td>00A7</td>
<td>3E00</td>
<td>181</td>
<td>LD</td>
<td>A, DRTWO</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00A9</td>
<td>D307</td>
<td>182</td>
<td>OUT</td>
<td>(DRTCB), A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00AB</td>
<td>3E30</td>
<td>183</td>
<td>LD</td>
<td>A, SRCRES</td>
<td>RESET SRC</td>
<td></td>
</tr>
<tr>
<td>00AD</td>
<td>D307</td>
<td>184</td>
<td>OUT</td>
<td>(DRTCB), A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00AF</td>
<td>3A4020</td>
<td>185</td>
<td>LD</td>
<td>A, (DRTFLG) ; UPDATE FLAG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00B2</td>
<td>CBE5</td>
<td>186</td>
<td>SET</td>
<td>5, A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00B4</td>
<td>324020</td>
<td>187</td>
<td>LD</td>
<td>(DRTFLG), A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00B7</td>
<td>C9</td>
<td>188</td>
<td>RET</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00B8</td>
<td>CDF900</td>
<td>189</td>
<td>CALL</td>
<td>SAVE</td>
<td>,CH. A TX BUFFER EMPTY</td>
<td></td>
</tr>
<tr>
<td>00BA</td>
<td>2A4120</td>
<td>190</td>
<td>LD</td>
<td>HL, (BUF PTR) ; GET BUFFER PTR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00BE</td>
<td>46</td>
<td>191</td>
<td>LD</td>
<td>B, (HL) ; GET CHAR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00BF</td>
<td>7D</td>
<td>192</td>
<td>LD</td>
<td>A, L ; UPDATE PTR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00C0</td>
<td>3C</td>
<td>193</td>
<td>INC</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00C1</td>
<td>E63F</td>
<td>194</td>
<td>AND</td>
<td>3FH ; 164 BYTE WRAPAROUND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00C3</td>
<td>6F</td>
<td>195</td>
<td>LD</td>
<td>L, A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00C4</td>
<td>24120</td>
<td>196</td>
<td>LD</td>
<td>(BUF PTR), HL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00C7</td>
<td>7B</td>
<td>197</td>
<td>LD</td>
<td>A, B ; OUTPUT CHAR.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00CB</td>
<td>D304</td>
<td>198</td>
<td>OUT</td>
<td>(DRTDA), A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00CA</td>
<td>C9</td>
<td>199</td>
<td>RET</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00D2</td>
<td>C9</td>
<td>200</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00D3</td>
<td>C9</td>
<td>201</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00D4</td>
<td>C9</td>
<td>202</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00D5</td>
<td>C9</td>
<td>203</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00D6</td>
<td>C9</td>
<td>204</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00D7</td>
<td>C9</td>
<td>205</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00D8</td>
<td>C9</td>
<td>206</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00D9</td>
<td>C9</td>
<td>207</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00DA</td>
<td>C9</td>
<td>208</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00DB</td>
<td>C9</td>
<td>209</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00DC</td>
<td>C9</td>
<td>210</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00DD</td>
<td>C9</td>
<td>211</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00DE</td>
<td>C9</td>
<td>212</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00DF</td>
<td>C9</td>
<td>213</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>CDF900</td>
<td>214</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0002</td>
<td>CDF900</td>
<td>215</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0003</td>
<td>CDF900</td>
<td>216</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0004</td>
<td>CDF900</td>
<td>217</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

751-1809-0001  2-93  2-18-81
MATHEWS SAVE REGISTER ROUTINE

00F9 E3 239 EX (SP),HL ; SP = HL
00FA D5 240 PUSH DE ; DE
00FB C5 241 PUSH BC ; BC
00FC F5 242 PUSH AF ; AF
00FD DDE5 243 PUSH IX ; IX
00FE FDE5 244 PUSH IY ; IY
0101 C0DF01 245 CALL QO ; PC
0104 FDE1 246 POP IY
0106 DDE1 247 POP IX
0108 F1 248 POP AF
0109 C1 249 POP BC
010A D1 250 POP DE
010B E1 251 POP HL
010C FB 252 EI
010D ED4D 253 RETI
254
255 QQ.
256 JP (HL)
257 "E
258
259 ;, CONSTANTS
260
261 DRTTA
0110 00 262 DEFB DRTWRO ; CHAN. RESET
0111 18 263 DEFB CHRES
0112 01 264 DEFB DTRWR1 ; CHAN. CHARACS.
0113 13 265 DEFB RXIAP+TXI+EXTI
0114 04 266 DEFB DTRWR4 ; MODE
0115 4F 267 DEFB X16+STOP2+EVEN+PARITY
0116 05 268 DEFB DTRWR5 ; TX PARAMS.
0117 AA 269 DEFB DTR+TX7+TXEN+RTS
0118 4F 270 DEFB DTRWR3 ; RX PARAMS.
0119 41 271 DEFB RX7+RXEN
0120 75 272 DRTEA. EQU $4
273
274 DRTTB
011A 00 275 DEFB DRTWRO ; CHAN. RESET
011B 18 276 DEFB CHRES
011C 01 277 DEFB DTRWR1 ; CHAN. CHARACS.
011D 17 278 DEFB RXIAP+DRTSAV+TXI+EXTI
011E 02 279 DEFB DTRWR2 ; VECTOR REG
011F 10 280 DEFB DRTVEC. AND. 255
0120 04 281 DEFB DTRWR4 ; MODE
0121 4F 282 DEFB X16+STOP2+EVEN+PARITY
0122 05 283 DEFB DTRWR5 ; TX PARAMS.
0123 AA 284 DEFB DTR+TX7+TXEN+RTS
0124 03 285 DEFB DTRWR3 ; RX PARAMS.
0125 41 286 DEFB RX7+RXEN
287   DRTEB  EQU  $0
288
289
0126  2B  290   DEFB  2BH ; PORT B MODE
0127  00   291   DEFB  0000000B ; DATA DIRECTION
0128  2B  292   DEFB  2BH
0129  EE   293   DEFB  11101110B ; " " " PORT C
012A  06   294   DEFB  6
012B  0E   295   DEFB  00001110B ; ACT1 MODE
012C  1C   296   DEFB  10H ; ACT1 MODE
012D  02   297   DEFB  11000010B ; ACT2 MODE
012E  1D   298   DEFB  1DH
012F  C2   299   DEFB  11000010B ; ACT3 MODE
0130  1E   300   DEFB  1EH
0131  C2   301   DEFB  11000010B
0132  16   302   DEFB  16H ; ACT1 TC MSB
0133  00   303   DEFB  0
0134  17   304   DEFB  17H ; LSB
0135  06   305   DEFB  C10CNT
0136  18   306   DEFB  18H ; ACT2 TC MSB
0137  00   307   DEFB  0
0138  19   308   DEFB  19H ; LSB
0139  06   309   DEFB  C10CNT
013A  1A   310   DEFB  1AH ; ACT3 TC MSB
013B  00   311   DEFB  0
013C  18   312   DEFB  1BH ; LSB
013D  06   313   DEFB  C10CNT
013E  01   314   DEFB  1 ; MASTER CONFIG. REG.
013F  F0   315   DEFB  11110000B ; ACT1 TRIGGER
0140  0A   316   DEFB  10
0141  06   317   DEFB  00001110B ; ACT2 TRIGGER
0142  0B   318   DEFB  11 ; ACT2 TRIGGER
0143  06   319   DEFB  00001110B ; ACT3 TRIGGER
0144  0C   320   DEFB  12
0145  06   321   DEFB  00001110B
322   CEND:  EQU  $0
323 *E
324

DATA AREA

2000  2000  327  ORG  RAM
2000
2040  2040  328  BUFFER:  DEFS  64
2041  2041  329  DRTFLG:  DEFS  1
2043  2043  330  BUFPTR:  DEFS  2
331  331  DEFS  64 ; STACK AREA
332  STACK:  EQU  $0
333
334   END
There are several differences between the 8500 devices and the Z80 family peripheral devices, including interrupt handling, reset to the device, and daisy-chain control.

This application brief describes the hardware interface requirements and interrupt structure of the 8500 series peripherals in Z80 systems. The 8500 peripherals are general-purpose versions of the Z-BUS counterparts and are designed to interface to nonmultiplexed buses (such as in a Z80 system), instead of multiplexed buses (such as in the Z8000).

### CPU HARDWARE

The hardware interface consists of three basic groups of signals: the data bus, control and selection lines, and the interrupt control lines. Following is a table of the general interface signals used by the CPU. Additional information can be found in the peripherals' separate data sheets.

#### DATA BUS

- **D<sub>0</sub>-D<sub>7</sub>** Data bus, bidirectional, 3-state. This bus is used to transfer data between the CPU and the peripheral device.

#### CONTROL SIGNALS

- **A<sub>0</sub>-A<sub>n</sub>** Address select lines (optional). These lines are normally used to select the port and/or control registers.
- **CE** Chip Enable. CE should be gated with TORQ or MREQ to prevent spurious chip selects during other machine cycles.

#### INTERRUPT OPERATION

Understanding the 8500 interrupt operation requires basic operational knowledge of the Interrupt Pending (IP) and Interrupt Under Service (IUS) bits in relation to the daisy chain. IP is set in the SIO by an interrupt condition, such as the transmit buffer going empty, and is used with IUS to control the INT signal. IP is not set while the CPU is executing an interrupt acknowledge cycle. Thus,

\[ IP = INT \land \neg VREAD \]

The IP latch is cleared either by a software command to the device or by an implicit action generated by the interrupt service routine. The implicit action may be triggered by the CPU reading or writing a register in the device. For example, on a serial receive device like the SIO, IP may be reset when the CPU reads the character from the receive buffer that caused the interrupt. This removes the interrupt condition, allowing other interrupts to occur.

The Interrupt Under Service (IUS) latch is used to designate the interrupt that is
currently being serviced. IUS is set when the device receives an interrupt acknowledge from the CPU while IEL is High and IP is set. If IEL is Low, the device is prevented from setting the IUS latch and thus cannot issue a vector. In this way, the daisy chain can establish relative priority among peripheral devices. IUS is cleared on the 8500 devices by an explicit software command.

The daisy chain used in the Z80 peripherals is referred to as an IP and IUS daisy chain, because the IP and IUS bits control the IEO pin and the lower portion of the chain. If IP is set, IEO can be Low even if another peripheral has an interrupt under service. When the CPU executes an RETI instruction (ED-4D opcode), the peripheral monitors the bus and resets IUS. If the CPU reads the "ED" part of RETI, peripherals with IP set and IEL High bring IEO High momentarily. This enables the device in the chain with IUS set to clear its IUS latch when the "4D" byte is read by the CPU. (IUS for a device is not cleared unless IEL is High and the "ED-4D" instruction is decoded. This allows more than one device to have IUS set so that nested interrupts can be implemented.)

On the 8500 series devices, IP is used to control the daisy chain only during the interrupt acknowledge cycle. Under normal conditions, only IUS is required to control the state of the IEO pin. Therefore, the daisy chain used in 8500 devices is referred to as an IUS daisy chain. Since IP is not a part of the daisy chain, there is no "ED" decoding pulling IEO High when IP is set. To allow more control over the daisy chain, the 8500 devices have a "Disable Lower Chain" (DLC) software command that unconditionally brings IEO Low. This can be used to deactivate parts of the daisy chain selectively, regardless of interrupt status. Figure 1 shows the functions of IP and IUS and the truth tables for each.

A unique feature of the 8500 devices is the INTACK pin. This pin acknowledges a CPU interrupt service cycle to the peripheral, allowing the peripheral to gate its vector onto the data bus. On the Z80 peripherals, interrupt acknowledge cycles from the CPU consist of a special M1 cycle where LORQ is activated instead of MREQ. This limits the control of devices in systems using a processor other than the Z80. As a result, a simpler implementation has been devised, which uses additional logic to accommodate a wider variety of processors. Figure 2 shows a circuit that generates INTACK for the 8500 devices in addition to wait states. Figure 3 shows the timing for INTACK and wait generation.

---

**Figure 1. 8500 Device Interrupt-Processing Sequence**

---

**Figure 2. INTACK and WAIT Generation for 8500 Peripherals**
Figure 3. Timing for 8500 Peripherals During Interrupt Acknowledge Without Z80 Peripheral Logic

On long daisy chains, wait states may be necessary to allow the IEO and IEI lines time to stabilize, thus avoiding conflict between devices and preventing IUS or IP from changing erroneously. Because of the IP and IUS configurations, the daisy chain used in Z80 peripherals needs to stabilize during the Interrupt acknowledge and RETI operations.

However, on the 8500 devices, the daisy chain is IUS and wait states are generated for the INTACK cycle only, not for the return cycle. (There is no "ED-4D" decode.) As a result, hardware interfacing is greatly simplified and timing is less complicated than on the Z80 peripherals.

SOFTWARE CONSIDERATIONS

There are several options available for servicing interrupts on the 8500 devices. Since the vector register (or IP register) can be read at any time, the software can emulate the Z80 CPU interrupt response easily. The interrupt vector reflects the interrupt status condition, even if the peripheral is programmed to return a vector that does not reflect the status change (SAV or VIS not set). This allows a simple software routine to emulate the Z80 vector response operation, as shown in the code of Figure 4.

```
<table>
<thead>
<tr>
<th>Loc.</th>
<th>Obj Code</th>
<th>M</th>
<th>Stmt Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>3E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>3000</td>
<td>20</td>
<td>INDEX: LD A,CIVR; CURRENT INT. VECTOR REG</td>
</tr>
<tr>
<td>0002</td>
<td>0800</td>
<td>21</td>
<td>OUT CCTRL; WRITE REL. PTR.</td>
</tr>
<tr>
<td>0004</td>
<td>0800</td>
<td>22</td>
<td>IN A,CCTRL; READ VECTOR REG.</td>
</tr>
<tr>
<td>0006</td>
<td>3C</td>
<td>23</td>
<td>INC A; VALID VECTOR</td>
</tr>
<tr>
<td>0007</td>
<td>C8</td>
<td>24</td>
<td>RET Z; NO INTERRUPT - RETURN</td>
</tr>
<tr>
<td>000B</td>
<td>650E</td>
<td>25</td>
<td>AND 00001100; MASK OTHER BITS</td>
</tr>
<tr>
<td>000A</td>
<td>5F</td>
<td>26</td>
<td>LD E,A; FORM INDEX VALUE</td>
</tr>
<tr>
<td>0008</td>
<td>1600</td>
<td>27</td>
<td>LD D,0</td>
</tr>
<tr>
<td>0000</td>
<td>216000 R</td>
<td>28</td>
<td>LD HL,VECTAB; ADD VECTOR TABLE ADDR</td>
</tr>
<tr>
<td>0010</td>
<td>19</td>
<td>29</td>
<td>ADD HL,DL</td>
</tr>
<tr>
<td>0011</td>
<td>7E</td>
<td>30</td>
<td>LD A,H; GET LOW BYTE</td>
</tr>
<tr>
<td>0012</td>
<td>23</td>
<td>31</td>
<td>INC HL</td>
</tr>
<tr>
<td>0013</td>
<td>66</td>
<td>32</td>
<td>LD H,(HL); GET HIGH BYTE</td>
</tr>
<tr>
<td>0014</td>
<td>6F</td>
<td>33</td>
<td>LD A,L; PUT ROUTINE ADDR IN HL</td>
</tr>
<tr>
<td>0015</td>
<td>19</td>
<td>34</td>
<td>JP (HL); GO TO ROUTINE 1</td>
</tr>
<tr>
<td>35</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0016</td>
<td>0010</td>
<td>37</td>
<td>DEFN INT1</td>
</tr>
<tr>
<td>0018</td>
<td>0011</td>
<td>38</td>
<td>DEFN INT2</td>
</tr>
<tr>
<td>001A</td>
<td>0012</td>
<td>39</td>
<td>DEFN INT3</td>
</tr>
<tr>
<td>001C</td>
<td>0013</td>
<td>40</td>
<td>DEFN INT4</td>
</tr>
<tr>
<td>001E</td>
<td>0014</td>
<td>41</td>
<td>DEFN INT5</td>
</tr>
<tr>
<td>0020</td>
<td>0015</td>
<td>42</td>
<td>DEFN INT6</td>
</tr>
<tr>
<td>0022</td>
<td>0016</td>
<td>43</td>
<td>DEFN INT7</td>
</tr>
<tr>
<td>0024</td>
<td>0017</td>
<td>44</td>
<td>DEFN INT8</td>
</tr>
</tbody>
</table>
```

Figure 4. Z80 Vector Interrupt Response Emulation by Software
A SIMPLE Z80 SYSTEM

The 8500 devices interface easily to the Z80 CPU, providing a system of considerable flexibility. Figure 5 illustrates a simple system using the Z80 CPU and a Z8536 CIO in a noninterrupt environment. Since INTACK is not used, it is tied High and no additional logic is needed. Because the CIO can be used in a polled interrupt system, the INT pin is connected to the CPU. The Z80 should not be programmed for Interrupt Mode 2, because the vector from the CIO is never sent to the CPU. Instead, the CPU can be set for Interrupt Mode 1, and a global interrupt routine that reads the vector register from the CIO can determine which routine to go to when an interrupt occurs, as previously illustrated in Figure 4.

Figure 5. Non-Interrupt CPU Interface

A Z80 system using a combination of Z80 family peripherals and 8500-type peripherals is easily constructed, as shown in Figure 6. There is no placement restriction on the 8500 devices within the daisy chain, but it is recommended that they be near the beginning of the chain in order to minimize propagation delays during the "ED-4D" decoding. The 8500 devices do not decode the "ED" during an opcode fetch cycle, so IE0 will not change state during this time.

NOTE. Z80 DMA uses the WR line also.

Figure 6. A Z80 System Using 8500 Devices and Z80 Peripherals

Figure 7 is a diagram of the logic represented by the WAIT and INTACK logic box in Figure 6. The WAIT signal is OR-wired to the output of each peripheral device (if used). The RD and WR signals only go to the 8500 device. The Z80 peripherals are wired to the Z80 as usual. The timing for the INTACK and WAIT generation logic is illustrated in Figure 8.
Figure 7. WAIT and INTACK Generation Logic

Figure 8. Timing for 8500 and Z80 Peripherals During Interrupt Acknowledge
When an external clock is not provided in a Z80-based system, it is often necessary to generate a bit-rate clock for serial devices. The most efficient way to accomplish this is to use a programmable counter that can change the bit-rate clock under CPU control. In this example, the Z8536 Counter/Timer I/O device (CIO) was chosen to generate the bit-rate clocks for a Z80-based statistical multiplexer project that used a Z80 S10 and a Z80 DART.

This application brief describes the use of the Z8536 CIO device in a Z80-based system for generating the bit-rate clocks for asynchronous communications. The Z8536 CIO contains the circuitry necessary to generate the clock pulses required by asynchronous communication devices.

The Z8536 CIO is housed in a 40-pin package and contains both system bus interface and I/O port connections. The three 16-bit counters can be programmed to output a pulse, square wave, or one-shot waveform on the timer's corresponding output pin. Three bits of the output ports (two from Port B and one from Port C) are used as the counter/timer outputs and provide the bit-rate pulses used in this application.

Interfacing the CIO to the Z80 CPU requires eight bidirectional data lines and five control lines. The data lines are used to transfer register address and data to or from the CIO via the RD, WR, CE, and address control lines. Two address lines (A0 and A1) select the port the CPU is accessing. Table 1 shows the port selected by the address bits.

<table>
<thead>
<tr>
<th>Address Line</th>
<th>A1</th>
<th>A0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port C</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Port B</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Port A</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>CTRL</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The control port (CTRL) is used for control register selection and parameter transfer. To select a particular register, a Register Pointer is written to the CTRL port and the data is written into or read from the register.

The CIO contains a state machine that controls the CPU interface. Upon power-up, the CIO is placed in a reset state and remains there until cleared by the program. Reset can also be initiated by issuing a command to Register 0 with bit 0 set or by a hardware condition (RD and WR simultaneously active). The reset state is described in detail in the programming section. Once the reset state is cleared, the CIO is placed in state 0, in which the control registers can be accessed by writing a Register Pointer to the CIO control port. This places the CIO in state 1, after which the next CPU access (read or write register data) causes the CIO to revert to state 0. The last register addressed may be accessed simply by reading the CIO control port. It should be noted that the Register Pointer can be written only while in state 0. Also, data can be written to a control register only after a Register Pointer has been written. Figure 1 shows the state diagram for the CIO.

The RD and WR control lines determine the data path direction into or out of the CIO. When activated simultaneously, they also perform the device's reset function. Figure 2 illustrates how the reset function can be implemented using external circuitry.
Since interrupts are not used in this application, INTACK is tied High to prevent spurious interrupt operation of the C10 due to noise.

Each counter/timer uses one or more bits on one of the parallel ports to provide for counter input and counter/timer output. Table 2 shows which output port bits correspond to particular counter/timer inputs and outputs.

The outputs of the counter/timers (PB4, PBO, and PCO) are fed to the rest of the circuitry to supply the serial clock pulses.

![Diagram](image_url)

**Figure 2. RESET Interface to the Z8536**

**Table 2. Counter/Timer External Interface Bits**

<table>
<thead>
<tr>
<th>Function</th>
<th>C/T1</th>
<th>C/T2</th>
<th>C/T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>C/T Output</td>
<td>PB4*</td>
<td>PB0</td>
<td>PC0</td>
</tr>
<tr>
<td>Counter Input</td>
<td>PB5</td>
<td>PB1</td>
<td>PC1</td>
</tr>
<tr>
<td>Trigger Input</td>
<td>PB6</td>
<td>PB2</td>
<td>PC2</td>
</tr>
<tr>
<td>Gate Input</td>
<td>PB7</td>
<td>PB3</td>
<td>PC3</td>
</tr>
</tbody>
</table>

*PB4 = Port B, bit 4

The last hardware consideration involves the clock input, PCLK. Since the Z8536 does not need to be synchronized with the CPU clock, PCLK can come from any source so long as it meets the timing and interface requirements. In fact, PCLK can come from a source external to the system if desired. Once inside the device, PCLK is divided by two before it is sent to the counter/timer circuits. There is no other prescaling done and the resulting clock is fed to the 16-bit counters.

The configuration of the C10 defines the general operating characteristics of the device with respect to its internal functions. The Port Mode Specification register sets to output those bits in Port B that are used for the counter/timer outputs. In this example, Bit mode is used on Ports B and C to output the counter/timer pulses.

The Counter/Timer mode, time constant values, and trigger commands are the last parameters to be set. Finally, the Master Configuration Control register is set to enable Port B, all the counter/timers, and Port C (Port C is enabled along with the counter/timers). The Counter/Timer mode is programmed for continuous cycle square wave with external output enabled. The square-wave cycle time is two times the programmed time constant, which must be taken into account when programming time constant values. The downcounters in the C10 are 16-bit counters that are decremented by one for each internal clock cycle. The internal clock cycle is the PCLK cycle divided by two, so the time constant value is determined by the following formula:

\[
\text{Time Constant} = \frac{\text{PCLK}}{4 \times \text{Output Frequency}}
\]

PCLK is divided by four in the formula because it is divided by two inside the C10 before being fed into the downcounter and by two again because a square wave cycle is two times the time constant value. Substituting the baud rate and a multiplier of 16 for the output frequency, the formula reduces to a simple time constant formula.

\[
TC = \frac{\text{PCLK}}{4 \times 16 \times \text{Baud Rate}}
\]

With a 3,686.4 MHz PCLK input and a desired 9600 baud rate, the formula simplifies to:

\[
TC = \frac{3686400}{4 \times 16 \times 9600} = \frac{3686400}{57600} = 6
\]

Other 16X baud rates may be generated by using the above formula in a general form.

\[
TC = \frac{3686400}{4 \times 16 \times \text{Baud Rate}}
\]

The user must exercise caution when choosing values for the PCLK and baud rates since they must result in nearly integral time constant values. For example, a 2,4576 MHz clock input with 9600 baud and a 16X clock output give a time constant value of 4. Greater flexibility is available for selecting time constant values because the SIO does not require a square wave input when programmed for 16X, 32X, or 64X clock inputs. Pulses may be used with the SIO provided the user adheres to the SIO timing requirements.

The last operation performed on the C10 is a trigger command to "Kick it off." This also includes setting the gate command bit in the Counter/Timer Command and Status registers, which allows the clock pulses to toggle the
CONCLUSION

The designer should find the Z8536 CIO a versatile and cost-effective component to satisfy his or her system needs. Coupled with other Zilog components, the Z8536 architecture enhances the performance of any Z80 system by providing the essential timing, I/O functions, and interrupt control functions necessary for efficient system operation.

The Z8536 CIO was chosen after considering device count, performance, and ease of use. Alternatives to the CIO include discrete (TTL) hardware counters and gates, external clock sources, or the Z80 CTC. These methods are generally too parts-intensive, and power consumption is therefore higher. For applications where two 8-bit ports and three counter/timers are needed, the CIO proves to be the ideal component.

APPENDIX

Following is a listing of a test program written for the Z80 CPU. This program simply initializes the CIO and then loops until stopped, with the CIO continuously providing pulses. All three counter/timers are used to generate square waves corresponding to a 16X 9600 baud clock.

```
LOC   OBJ CODE M STMT SOURCE STATEMENT
 1   ; CIO TEST PROGRAM
 2 ;[11] 01-07-81/MDP INITIAL CREATION
 3
 4 ; THIS PROGRAM Initializes THE THREE COUNTER
 5 ; TIMERS IN THE Z8536 CIO TO GENERATE SQUARE
 6 ; WAVES. THEN LOOPS FOREVER.
 7
 8 ; PROGRAM EQUATES
 9
10  CIOC: EQU B ; CIO PORT C
11  CIOB: EQU CIOC+1 ; CIO PORT B
12  CIOA: EQU CIOC+2 ; CIO PORT A
13  CIOCTL: EQU CIOC+3 ; CIO CTRL PORT
14  BAUD: EQU 9600 ; ASYNC BAUD RATE
15  RATE: EQU BAUD/100
16  CIOCNT: EQU 576/RATE
17  RAM EQU 2000H ; RAM START ADDR
18  RAMSIZE EQU 1000H ; RAM SIZE
19  *E
20
21 ; *** MAIN PROGRAM ***
22
23  ORG 0
24 BEGIN:
25  LD SP,STAK ; INIT SP.
26  CALL INIT ; INIT DEVICES
27
28  JR */ LOOP FOREVER:
29
30  INIT:
31  CIOINI:
32  IN A,(CIOCTL) ; INSURE STATE 0
33  LD A,0 ; REG 0 OR RESET
34  OUT (CIOCTL),A ; WRITE PTR OR CLEAR RESET
35  IN A,(CIOCTL) ; STATE 0
36  LD A,0 ; REG 0
37  OUT (CIOCTL),A ; WRITE PTR
38  LD A,1 ; WRITE RESET
39  OUT (CIOCTL),A ; CLEAR RESET
40  LD A,0 ; CLEAR RESET
41  OUT (CIOCTL),A ; INIT CIO
42  LD HL,CLST ; INIT CIO
43  LD B,CEND-CLST
44  LD C,CIOCTL
45  DTIR
46  RET
47 *E
48
```
TEST.CIO

LOC OBJ CODE M STMT SOURCE STATEMENT ASM 5.9

49 ;; CONSTANTS
50
51 CLST:
0026 28
0027 00
0028 2B
0029 EE
002A 06
002B FE
002C 1C
002D C2
002E 1D
002F C2
0030 1E
0031 C2
0032 16
0033 00
0034 17
0035 06
0036 1B
0037 00
0038 19
0039 06
003A 1A
003B 00
003C 1B
003D 06
003E 01
003F F0
0040 0A
0041 06
0042 0B
0043 06
0044 0C
0045 06
0046
0047
0048
0049
004A
004B
004C
004D
004E
004F
0050
0051
0052
0053
0054
0055
0056
0057
0058
0059
005A
005B
005C
005D
005E
005F
0060
0061
0062
0063
0064
0065
0066
0067
0068
0069
006A
006B
006C
006D
006E
006F
0070
0071
0072
0073
0074
0075
0076
0077
0078
0079
007A
007B
007C
007D
007E
007F
0080
0081
0082
0083
0084
0085
0086
0087
0088
0089
008A
008B
008C
008D
008E
008F
0090
0091
0092
0093
0094
0095
0096
0097
0098
0099
009A
009B
009C
009D
009E
009F
00A0
00A1
00A2
00A3
00A4
00A5
00A6
00A7
00A8
00A9
00AA
00AB
00AC
00AD
00AE
00AF
00B0
00B1
00B2
00B3
00B4
00B5
00B6
00B7
00B8
00B9
00BA
00BB
00BC
00BD
00BE
00BF
00C0
00C1
00C2
00C3
00C4
00C5
00C6
00C7
00C8
00C9
00CA
00CB
00CC
00CD
00CE
00CF
00D0
00D1
00D2
00D3
00D4
00D5
00D6
00D7
00D8
00D9
00DA
00DB
00DC
00DD
00DE
00DF
00E0
00E1
00E2
00E3
00E4
00E5
00E6
00E7
00E8
00E9
00EA
00EB
00EC
00ED
00EE
00EF
00F0
00F1
00F2
00F3
00F4
00F5
00F6
00F7
00F8
00F9
00FA
00FB
00FC
00FD
00FE
00FF

52 DEFB 28H ; PORT B MODE
53 DEFB 0000000B ; PORT B DIRECTION
55 DEFB 11101110B ; PORT C DIRECTION
56 DEFB 06H ; PORT C DIRECTION
57 DEFB 11111110B ; CT1 MODE
58 DEFB 1CH ; CT1 MODE
59 DEFB 11000010B ; CT2 MODE
60 DEFB 1DH ; CT2 MODE
61 DEFB 11000010B ; CT3 MODE
62 DEFB 1EH ; CT3 MODE
63 DEFB 11000010B ; CT3 MODE
64 DEFB 16H ; CT1 TC MSB
65 DEFB 0 ; PORT C DIRECTION
66 DEFB 17H ; LS8
67 DEFB CI0CNT ; LS8
68 DEFB 18H ; CT2 TC MSB
69 DEFB 0 ; PORT C DIRECTION
70 DEFB 19H ; LS8
71 DEFB CI0CNT ; LS8
72 DEFB 1AH ; CT3 TC MSB
73 DEFB 0 ; PORT C DIRECTION
74 DEFB 18H ; LS8
75 DEFB CI0CNT ; LS8
76 DEFB 1 ; MASTER CONFIG. REG.
77 DEFB 11110000B ; CT2 TC MSB
78 DEFB 0AH ; TRIGGER
79 DEFB 00000110B ; PORT C DIRECTION
80 DEFB 08H ; TRIGGER
81 DEFB 00000110B ; PORT C DIRECTION
82 DEFB 0CH ; TRIGGER
83 DEFB 00000110B ; PORT C DIRECTION
84 CEND: EQU * ; DATA AREA
85
86
87
88 ORG RAM
89 DEFS 64 ; STACK AREA
90 STAK: EQU * ; STACK AREA
91
92 END
Timing in an Interrupt-Based System with the Z80® CTC

In many computer systems, an accurate time base is needed so that critically timed events do not go awry. Use of a counter or timer to monitor time-dependent activities is essential in such systems. In an interrupt-driven system, the Z80 CTC can provide regular program time intervals. Single-event counts or single-event time delays can also be implemented under program control. This application note describes both continuous time-interval operations and single-interval count operations using the Z80 CTC in a Z80 system.

In the example used here, the hardware consists of a Z80 CPU with 4K bytes of RAM, 4K bytes of ROM, a Z80A SIO, and a Z80A CTC. There are two external inputs to the CTC: one is derived from the ac power line to provide 60Hz pulses; the other is connected to a transmit clock line on the SIO. One of the counter/timer outputs is connected to the SIO transmit and receive clock input, as shown in Figure 1.

![Z80A System Block Diagram](image)

Figure 1. Z80A System Block Diagram

March 1981
The Z80 CTC is designed for easy interface to the Z80 CPU. An 8-bit bidirectional data bus is used to transfer information between the CTC and CPU. The control lines, RD, TORQ, MT, and CE, determine what data is being transmitted and when. MT and TORQ are used during the interrupt acknowledge cycle to allow the CTC to present its 8-bit interrupt vector to the CPU. TORQ is also used in conjunction with CE to enable transfers between the CTC and the CPU. RD is used to control the direction of data flow between the CTC and the CPU. The channel select lines (CS0 and CS1) are connected to the lowest two bits of the address bus and are used to access one of the four counter/timer channels. Table 1 shows the relationships between the CS pins and the counter/timer channels.

**Table 1. Channel Select Values**

<table>
<thead>
<tr>
<th>CS1</th>
<th>CS0</th>
<th>C/T Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Channel 0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Channel 1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Channel 2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Channel 3</td>
</tr>
</tbody>
</table>

The CTC system clock input requirements are similar to those of the Z80 CPU. For both, the system clock input low level should be less than 0.45 V, the high level should be no less than VCC - 0.6 V, and the clock rise and fall times should be less than 30 ns. A clock-driver device that meets these requirements, such as the HH-3006-A, works well with the CTC. Several devices can be connected to the driver, but the user should be careful not to overload the driver. The capacitance of the clock input to the CTC (20 pF) should be noted as this may affect the system clock rise and fall times.

Interrupt control logic within the CTC is used to initiate interrupts and to control the interrupt acknowledge cycle generated by the CPU. An interrupt is generated by the CTC when one of the counter/timer down counters reaches terminal count (0) and IE1 is High. IE1 and IE0 allow the CTC to operate within the Z80 interrupt daisy chain and to connect to the next higher-priority and next lower-priority devices in the chain, respectively. If there is no higher-priority device, IE1 is tied to +5 V.

The CTC internally prioritizes each counter/timer with respect to interrupt generation. This maximizes performance by resolving contention between channels that should two or more interrupt conditions occur simultaneously.

Table 2 shows the relative priority levels of each counter/timer channel within the CTC.

**Table 2. CTC Channel Interrupt Priority**

<table>
<thead>
<tr>
<th>Priority</th>
<th>Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highest</td>
<td>0</td>
</tr>
<tr>
<td>Lowest</td>
<td>3</td>
</tr>
</tbody>
</table>

CTC MODES

There are two basic modes under which the CTC can operate: Timer mode and Counter mode. Each mode has certain programmable characteristics that enable the CTC to be used in a wide variety of applications.

**TIMER MODE**

A typical use of the CTC in Timer mode is to provide regular, fixed-interval interrupts to the CPU used as a time-base reference to allocate the processor resources efficiently. For example, a multitasking system might have the processor execute a task for a given length of time and then interrupt execution of the program at one-second intervals to scan the task queue for higher-priority tasks. This system time interval can be provided by the CTC in Timer mode. In Timer mode, the CTC downcounter is decremented by the output of the prescaler, which is toggled by the system clock input. The prescaler has a programmable value of 16 or 256, depending on the condition of bit 5 in the channel control word (CCW). Thus, with a 4 MHz system clock fed into the CTC, a timer resolution of 4μs (prescaler count of 16) or 64μs (count of 256) is possible.

In the example shown, the interrupt interval is set to 8.33 ms, which is provided by the CTC with a 3.6664 MHz input clock, 256 prescaler value, and a time constant value of 120. The CTC interrupt service routine uses a software count of 120 to maintain a one-second system time interval. Each time the service routine is executed, the software count is decremented by 1. When the count reaches 0, a flag is set and the program pursues an appropriate course of action. Figure 2 shows the initialization and interrupt service routine coding for a CTC channel using the Timer mode.

Another use of CTC Timer mode operation is to implement a nonretriggerable one-shot using external circuitry. The digital approach to the one-shot provides a programmable time delay under CPU control and provides greater noise immunity than the more common analog delay circuits provide. Figure 3 shows a circuit that uses part of a 74LS02 package in addition to one CTC channel.

The trigger waveform should be positive-going and should meet the CTC setup time for the CLK/TRIG input. Also, the trigger High level time should be less than the CTC delay time in order to prevent the two 74LS02s from latching in the triggered state. An additional gate can be added to initialize the 74LS02 flip-flop to a defined state when the system is reset or else the software can pulse the timer output to set the flip-flop, as is done in this case. A third use of the Timer mode is to provide a bit rate clock for a serial transceiver device, such as the Z80 S10. The S10 can accept a 1x, 16x, 32x, or 64x bit rate clock input from an external source, and with a 16x, 32x, or 64x multiplier, the S10 can accept a pulse waveform input for the bit rate clocks, as long as the pulses meet the rise, fall, and hold time requirements of the S10. The CTC meets these requirements and can be connected directly to the S10 to provide the necessary bit rate clocks. Figure 4 shows the code needed to generate a bit rate clock for the S10.

1 A clock driver by Hybrid House, 1615 Ramuda La., San Jose, CA 95112.
With a 1x bit rate clock programmed into the SIO, a square-wave input must be supplied. This can be done by adding a flip-flop between the CTC and the SIO. The time constant value should be set to half the baud rate value, since the CTC output is divided in half by the flip-flop.

---

**Figure 2. Software for CTC Timer Mode Operation**

**Figure 3. Monostable Multivibrator Using the Z80 CTC**
TEST CTCO

CTC TEST PROGRAM

THIS PROGRAM USES THE CTC IN CONTINUOUS TIMER MODE. THE CTC COUNTS SYSTEM CLOCK PULSES AND INTERRUPTS EVERY 120 PULSES. THEN DECREMENTS A COUNT, THEN SWITCHES THE LED STATE WHEN THE COUNT REACHES ZERO.

PROGRAM EQUATES

CTC0 EQU 12 . CIC 0 PORT
CTC1 EQU CTC0+1 . CIC 1 PORT
CTC2 EQU CTC0+2 . CIC 2 PORT
CTC3 EQU CTC0+3 . CIC 3 PORT

LITE EQU 0EOH : LIGHT PORT
RAM EQU 2000H : RAM START ADDR
RAMSIZE EQU 1000H

TIME EQU 120 . COUNT VALUE

CTC EQUATES

CICW EQU 1

INTEN EQU 80H

CTRMODE: EQU 40H
P256 EQU 20H
RISED: EQU 10H
PSTRT: EQU 8
TCLOAD: EQU 4
RESET: EQU 2

*** MAIN PROGRAM ***

ORG 0
JP BEGIN

ORG $ AND OFFFOH OR 10H

INTVEC

DEFW ICTC0
DEFW ICTC1
DEFW ICTC2
DEFW ICTC3

BEGIN

LD SP,STAK ; INIT SP
LD A,INTVEC/256 ; VECTOR INTERRUPT MODE
LD (COUNT),A ; CLEAR DISPLAY
LD (CTCO),A ; CLEAR CTC
CALL INIT ; INIT DEVICES
LD A,TIME ; INIT TIMER VALUE
LD (DISP),A ; CLEAR DISPLAY BYTE
LD A,TIME ; INIT TIMER VALUE
LD (COUNT),A

INIT:

LD A,INTEN+P256+TCLOAD+RESET+CCW
OUT (CTCO),A ; SET CTC MODE
OUT (CTCO),A ; SET TIME CONSTANT
OUT (CTCO),A ; SET VECTOR VALUE
XOR A
LD (DISP),A ; CLEAR DISPLAY BYTE
LD A,TIME ; INIT TIMER VALUE
LD (COUNT),A

INTERRUPT SERVICE ROUTINE
Figure 4. Software for CTC Bit Rate Generator
A typical computer system often uses a time-of-day clock. In the United States, the 60 Hz power line provides an accurate time base for synchronous motor clocks. A computer system can take advantage of the 60 Hz accuracy by incorporating a circuit that feeds 60 Hz square waves into a CTC channel. With a time constant value of 60, the CTC generates an interrupt once every second, which can be used to update a time-of-day clock. The CTC is set to Counter mode and with a time constant value of 60, as shown in Figure 5.

The interrupt service routine does nothing more than update the time-of-day clock. A more sophisticated operating system kernel would use the CTC to check the task queue status. In synchronous data communications, it is often necessary to ensure that a flag or sync character separates two adjacent message packets. Since some serial controller devices have no way to determine the status of sync characters sent, the user must use time delays to separate messages with the appropriate number of sync characters. Typically, software or timer delays are used to provide the time necessary to allow the characters to shift out of the serial device. The disadvantage of using this method is that variable baud rates shift characters at variable times so a worst-case time must be allowed if the baud rate is not known. If the bit rate clock is supplied by the modem, as is normally the case, this problem becomes even more acute.

A solution to this problem is to use a counter to count the number of bits shifted out of the serial device. With the CTC tied to the transmit clock line of the serial device, the CTC can be programmed to delay a certain number of bits before the CPU sends another message. This solves all of the problems mentioned and simplifies the message-handling software. Figure 6 shows the program needed to achieve the counting function. Note
that the interrupt service routine disables the CTC, because the CTC is used only once with each message. Otherwise, the CTC would generate an interrupt each time the counter reached terminal count.

Figure 1 shows the hardware implementation of the character delay counter using the CTC.

![Diagram of the CTC and its modes]

Figure 5. Software for CTC Counter Mode

```
TEST CTC1
LOC OBJ CODE M STMT SOURCE STATEMENT

1      CTC TEST PROGRAM
2      THIS PROGRAM COUNTS EXTERNAL PULSES AND
3      CHANGES THE LED STATE EVERY 60 COUNTS
4      PROGRAM EQUATES
5
6      CTC-EQUATES
7
8      CTC0: EQU 12 ; CTC 0 PORT
9      CTC1: EQU CTC0+1 ; CTC 1 PORT
10     CTC2: EQU CTC0+2 ; CTC 2 PORT
11     CTC3: EQU CTC0+3 ; CTC 3 PORT
12     LITE: EQU OEOH ; LIGHT PORT
13     RAM  EQU 2000H ; RAM START ADDR
14     RAMBIZ EQU 1000H
15     COUNT EQU 60 ; COUNTER TIME CONSTANT
16
17
18      CTC EQUATES
19
20     CCW   EQU 1
21     INTEN EQU BOH
22     CTRMODE: EQU 40H
23     P256: EQU 20H
24     RISEDG EQU 10H
25     PSTRT EQU 8
26     TCLOAD: EQU 4
27     RESET EQU 2
```
TEST CTC1

LOC OBJ CODE M STMT SOURCE STATEMENT

28 *E
29
30 ; *** MAIN PROGRAM ***
31
0000 0000 C31800 32 ORG 0
33 JP BEGIN
34
0010 0010 3B00 35 ORG $.OFFFOH.OR.1OH
36 INTEC:
0010 0012 0014 3B00 37 DEFW ICTC0
0012 0014 3800 38 DEFW ICTC1
0014 0016 3800 39 DEFW ICTC2
0016 0018 3800 40 DEFW ICTC3
41
42 BEGIN:
43 LD SP,STAK ;INIT SP
0018 0018 314020 44 IM 2 ;VECTOR INTERRUPT MODE
0018 0018 4500 45 LD A,INTVEC/256 ;UPPER VECTOR BYTE
001D 001D 3E00 46 LD I,A
001D 001E 3E00 47 CALL INIT ;INIT DEVICES
001E 001F 3D700 48 EI ;ALLOW INTERRUPTS
001F 0020 18FE 49 JR $ ;LOOP FOREVER
0020 0021 50
51 INIT:
0021 0022 3EC7 52 LD A,INTEN+CTRMODE+TCLOAD+RESET+CCW
0022 0023 D30D 53 OUT (CTC1),A ;SET CTC MODE
0023 0024 3E3C 54 LD A,COUNT
0024 0025 D30D 55 OUT (CTC1),A ;GET TIME CONSTANT
0025 0026 3E10 56 LD A,INTVEC.AND 11111000B
0026 0027 D30C 57 OUT (CTC0),A ;GET VECTOR VALUE
0027 0028 AF 58 XOR A
0028 0029 324020 59 LD (DISP),A ;CLEAR DISPLAY BY!
0029 002A C9 60 RET
61
62 *E
63
64 INTERRUPT SERVICE ROUTINE
65
66 ICTC0.
67 ICTC2:
68 ICTC3:
0030 0031 0032 0033 0034 0035 0036 0037 C9 69 EI ;DUMMY ROUTINES
0037 0038 FB 70 RETI
0038 0039 ED4D 71
72 ICTC1:
0039 003A 003B 003C 003D 003E 003F 0040 CD4800 73 CALL SAVE ;SAVE REGISTERS
0040 0041 3A4020 74 LD A,(DISP) ;BLINK LITES
0041 0042 2F 75 CPL
0042 0043 324020 76 LD (DISP),A
0043 0044 D3E0 77 OUT (LITE),A
0044 0045 C9 78 RET
79
80 SAVE REGISTER ROUTINE
81
82 SAVE:
0045 0046 0047 0048 0049 004A 004B 004C 004D 004E 004F 0050 0051 0052 0053 0054 0055 0056 E9 83 EX (SP),HL
0056 0057 84 PUSH DE
0057 0058 85 PUSH BC
0058 0059 86 PUSH AF
0059 005A CD5600 87 CALL GO
005A 005B 88 POP AF
005B 005C 89 POP BC
005C 005D 90 POP DE
005D 005E 91 POP HL
005E 005F 92 EI
005F 0060 ED4D 93 RETI
94
95 GO.
0060 0061 96 JP (HL)
96
97 *E
98

751-1809-0005 2-114 4/1/81
Figure 6. Software for CTC Single-Cycle Use
TEST.CTC3

LOC   OBJ CODE M STMT SOURCE STATEMENT

1 ;   CTC TEST PROGRAM
2 ;
3 ;   THIS PROGRAM INITIALIZES CTC INTERRUPT VECTOR,
4 ;   THEN STARTS CTC 3, THEN WAITS FOR CTC 3 TO
5 ;   TERMINATE. AFTER TERMINATING, THE CTC INTERRUPT
6 ;   THE CPU AND ENTERS A SERVICE ROUTINE THAT SETS
7 ;   A PROGRAM FLAG TO INDICATE ZERO COUNT, AND
8 ;   RESETS CTC 3.
9 ;
10 ;   EQUATES
11 ;
12 RAM:  EQU 2000H ;RAM START ADDRESS
13 RAM$IZ: EQU 1000H ;RAM SIZE
14 CTC0: EQU 12 ;CTC 0 PORT
15 CTC1: EQU CTCO+1 ;CTC 1 PORT
16 CTC2: EQU CTCO+2 ;CTC 2 PORT
17 CTC3: EQU CTCO+3 ;CTC 3 PORT
18 COUNT: EQU 20 ;COUNT 20 PULSES
19 ;
20 ;   CTC PARAMETERS
21 ;
22 CCW: EQU 1 ;CTRL BYTE
23 INTEN: EQU 90H ;INTERR. ENABLE
24 CTRMODE: EQU 40H ;COUNTER MODE
25 P256: EQU 20H ;PRESCALE BY 256
26 RISEDG: EQU 10H ;START ON RISING EDGE
27 PSTRT: EQU 8 ;PULSE STARTS TIMING
28 TLOAD: EQU 4 ;TIME CONST. FOLLOWS
29 RESET: EQU 2 ;SOFTWARE RESET
30 *E
31 ;
32 ORG 0
33 ORG C31B00
34 JP BEGIN ;GD MAIN PROGRAM
35 ORG 310H .AND. OFFHOH OR 10H
36 INTVEC: ORG 4100
37 CTCVEC:
38 DEFW ICTCO
39 DEFW ICTC1
40 DEFW ICTC2
41 DEFW ICTC3
42 ;
43 ;   MAIN PROGRAM
44 ;
45 BEGIN:
46 LD SP,STAK ;INIT SP
47 LD A,INTVEC/256 ;INIT VECTOR REG.
48 LD I,A
49 IM 2 ;VECTORED INTERRUPT MC
50 LD A,CTCVEC.AND.11111000B
51 OUT (CTCO),A ;SETUP CTC VECTOR
52 LD A,1 ;SET FLAG BYTE
53 LD (FLAG),A
54 EI
55 ;
56 LOOP:
57 LD A,(FLAG) ;READ FLAG BYTE
58 BIT 0,A
59 JR Z,LOOP ;BRANCH IF NOT SET
60 RES 0,A ;CLEAR FLAG BYTE
61 LD (FLAG),A
62 LD A,INTEN+CTRMODE+RISEDG+TLOAD+1
63 OUT (CTC3),A ;LOAD CTC 3
64 LD A,COUNT
65 OUT (CTC3),A
66 JR LOOP
67 *E
68 ;
69 ;   INTERRUPT SERVICE ROUTINES FOR CTC
70 ;
71 ICTCO:
72 ICTC1:
CONCLUSION

The versatility of the Z80 CTC makes it useful in a myriad of applications. System efficiency and throughput can be improved through prudent use of the CTC with the Z80 CPU. Coupled with the powerful, vectored interrupt capabilities of the Z80 CPU, the CTC can be used to supply counter/timer functions to the CPU. This reduces software overhead on the CPU and significantly increases system throughput.
Interfacing 16-Pin Dynamic RAMS to the Z80A Microprocessor

TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTRODUCTION</td>
<td>2-121</td>
</tr>
<tr>
<td>16 PIN DYNAMIC RAM ADDRESSING</td>
<td>2-122</td>
</tr>
<tr>
<td>MEMORY REFRESH</td>
<td>2-124</td>
</tr>
<tr>
<td>ACCESS TIME</td>
<td>2-125</td>
</tr>
<tr>
<td>Z80A/Z80 - CPU TIMING CONSIDERATIONS</td>
<td>2-127</td>
</tr>
<tr>
<td>MEMORY CYCLE SELECTION</td>
<td>2-131</td>
</tr>
<tr>
<td>DYNAMIC RAM MEMORY ORGANIZATION</td>
<td>2-132</td>
</tr>
<tr>
<td>SLOW MEMORY INTERFACE</td>
<td>2-138</td>
</tr>
<tr>
<td>DESIGN EXAMPLE</td>
<td>2-140</td>
</tr>
<tr>
<td>CONCLUSIONS</td>
<td>2-146</td>
</tr>
</tbody>
</table>
This application note will present the major design considerations and a design example for interfacing the 16-pin dynamic RAM devices, both 4K and 16K, to the Z80 and Z80A microprocessors. These devices will be emphasized because they are fast becoming the favorite memory component for data storage in microprocessor based systems. The 16K RAM (Zilog 6116), in particular, with design improvements over the 4K devices, will substantially reduce memory cost by quadrupling memory density in a package that is pin compatible with the 4K RAM.

This application note assumes a basic understanding of the Z80A CPU and dynamic RAM elements. The reader is referred to selected specification sheets on the various 4K and 16K dynamic RAMS and to the following Zilog literature:

Z80A CPU Technical Manual, and

Z6116 16K Dynamic RAM Product Specification
INTRODUCTION

16-pin dynamic RAMs are increasingly being used as the memory component for data storage in microprocessor-based systems. Their main features are low cost per bit and high bit density. These features, coupled with a low stand-by power mode, TTL-compatible inputs and outputs, and simple upgrade from 4K to 16K systems, have made these devices an attractive alternate to 18- or 22-pin dynamic RAMs.

Now, however, the system designer has to be concerned with the interface requirements of 16-pin dynamic RAMs. The characteristics of this memory element requires that refreshing of the memory be performed at periodic intervals in order to retain the stored data. This, coupled with the requirement for multiplexing address lines, has been the main drawback to their use. A typical interface generally required 12 to 20 standard TTL devices and included timing generators, decode logic, multiplexer circuitry, refresh logic, and buffers.

The Zilog Z80A microprocessor has been designed to simplify this interface with built-in refresh logic. This allows totally transparent RAM refresh without the need for a refresh counter or its associated multiplexer. During each memory opcode fetch cycle, a dedicated line from the CPU (RFSH) is used to indicate that a refresh read of all dynamic memories should be performed. With RFSH in the true state (LOW), the lower seven bits of the address bus identify one ROW address to be refreshed. Before the next opcode fetch, this address will have been incremented to point to the next ROW address. Since it is only necessary to refresh the 'ROWS', a total of 64 refresh cycles will refresh an entire 4K RAM, or 128 refresh cycles for a 16K RAM. Z80A-CPU refreshing is automatically performed during a portion of the instruction fetch cycle which is used for internal processing. Thus, the effect of refreshing the RAM is totally transparent to program execution, preventing the necessity of stealing cycles or stopping the CPU as would otherwise be required.
16 PIN DYNAMIC RAM ADDRESSING

Each cell of a dynamic RAM array is arranged in a matrix. Selection of a unique bit location within this matrix in a 4K RAM element will require 12 address lines while the 16K device requires 14. For the 16-pin RAM device to accommodate these lines, it will be required to divide them into two groups; Row addresses and Column addresses (six each for the 4K RAM and seven each for the 16K RAM). Each group is applied to the RAM on the same input lines (Figure 1) through an external multiplexer and latched into the chip by applying two clock strobos in succession. The first clock, the Row Address Strobe (RAS), latches the Row address bits into the RAM (A0-A5 for the 4K, A0-A6 for the 16K). The second clock, the Column Address Strobe (CAS), latches the Column address bits, (A6-A11 for the 4K, A7-A13 for the 16K) into the RAM.

Each cell, therefore, is uniquely addressed by row and column. When RAS goes active, all of the cells in the selected row respond (there are 64 rows in the 4K RAM matrix and 128 rows in the 16K RAM matrix) and are gated to sense amplifiers where the logic level of each cell is discriminated, latched, and rewritten. CAS activates a column in the matrix (there are 64 columns in the 4K RAM matrix and 128 columns in the 16K RAM matrix) which uniquely identifies the cell in the row output and yields the required bit to the output buffer.

During refresh, the interface logic will enable the Row Address lines from the multiplexer. The CPU, with a true condition on the Refresh line (RFSH), will then present the address (A0-A7) of the Row to be refreshed, and activate the memory request line (MREQ) to initiate a memory cycle.
FIGURE 1. The pin assignments for 4K and 16K RAMs show identical functions for each, except Pin 13, which is used as a chip select in 4K RAMs and as the 7th multiplexed address line in the 16K RAM.
MEMORY REFRESH

When any row in a 16-pin dynamic RAM is actively cycled, all locations within that row are refreshed. To refresh the entire RAM, it is only necessary to perform a RAS only memory cycle (CAS is not required for a refresh sequence) at each of the 64 row addresses for the 4K device and 128 row addresses for the 16K device, every 2 milliseconds or less.

The Z80 CPU refreshes the memory more frequently than is necessary to meet the 2ms row refresh requirement. Under worst case conditions, no more than 19T states will separate opcode fetch cycles (the EX (SP),HL instruction is representative of the longest time between opcode fetches). Assuming this worst case period between opcode fetches and, therefore, refresh cycles, the following times for total refresh for both 4K and 16K RAMS at 2.5 MHZ and 4 MHZ are shown below:

<table>
<thead>
<tr>
<th>MEMORY SIZE</th>
<th>Z80-CPU 2.5 MHZ</th>
<th>Z80A-CPU 4.0 MHZ</th>
<th>NO. OF REQUIRED REFRESH CYCLES/2 mS</th>
</tr>
</thead>
<tbody>
<tr>
<td>4K</td>
<td>487 us (max)</td>
<td>304 us (max)</td>
<td>64</td>
</tr>
<tr>
<td>16K</td>
<td>974 us (max)</td>
<td>608 us (max)</td>
<td>128</td>
</tr>
</tbody>
</table>

TABLE 1. WORST CASE MEMORY REFRESH CYCLES ASSUMING NO WAIT STATES

From the above table, it can be seen that the worst case refresh time for 16K RAMS consumes approximately 1/2 of the available 2ms time interval while the 4K RAM consumes only about 1/4 of the allotted time. This provides for optional use of the refresh cycle for other CPU transparent bus activity, such as DMA and CRT refresh.
ACCESS TIME

Most dynamic RAMS have access times in the range of 150ns to 300ns. This access begins with the leading edge of the row address strobe (RAS). The column address strobe (CAS) completes this access cycle. The time between the fall of RAS and the fall of CAS is identified as the RAS to CAS delay time (tRCD), and can be related to the previous access times as follows:

\[ \text{trACmax} = \text{trCDmax} + \text{tCACmax} \]

WHERE \( \text{trACmax} = \text{Access time from RAS} \)
\( \text{trCDmax} = \text{max RAS to CAS delay time} \)
\( \text{tCACmax} = \text{Access time from CAS} \)

As long as tRCD is less than max value (but greater than tRCDmin), the worst case access is from RAS (see Figure 1). If CAS is applied at a point in time beyond the trCDmax limit, the access time from RAS will be lengthened by the amount that tRCD exceeds the trCDmax limit and the access time from CAS (tCAC) will be the critical parameter. Note, however, that reducing tRCD to something less than trCDmax will have no effect at reducing trACmax.

The significance of the min/max value on tRCD is that CAS can be brought low any time within this window and not affect access time. This is a great improvement from early 4K designs that required CAS to be brought low at a set minimum time from RAS low in order to avoid increasing access time. This made no allowance for the time required to switch the MUX from ROW to COLUMN addresses, requiring that the worst case multiplexing time delay be added to the specified access time.

This window, for the application of the external CAS, is the result of gating CAS internal to the chip. The internal CAS is inhibited until the occurrence of a delayed signal derived from RAS. Therefore, CAS can be activated as soon as the requirement for the row address hold time (tRAH) has been satisfied and the address inputs have been changed from row to column. Note that the column address set-up time (tASC) can be assumed to be zero for all dynamic RAMs (See Figure 2).
Figure 2 Dynamic ram access time parameters
The Z80A/Z80 CPU is designed to allow efficient and effective interface with dynamic RAM memories. Figures 3 through 8 identify the timing for CPU data, address signals, and control signals associated with memory interface for the Z80A and Z80. The opcode fetch, with its associated refresh cycle, will represent the worst case memory access, requiring data to be returned to the CPU in the first two T states. Memory read and write cycles have relaxed timing requirements as indicated in Figures 5 through 8. This will require memories with access times of 250ns or less for the Z80A and 400ns or less for the Z80. These numbers, however, do not take into consideration the propagation delays through any buffer logic added.

Notice that addresses are stable well before MREQ goes active, giving sufficient time for address decode logic to settle. The main concern, therefore, is propagation delay from MREQ to RAS. This should be kept to a minimum since it will directly affect access time.

From Figure 7, it can be seen that write (WR) goes active on the trailing edge of T2. The CPU, therefore, usually performs a read-modify-write cycle (CAS before WR). To utilize the early write cycle (WR active before CAS) and allow 16K systems to tie their inputs and outputs together, the read line (RD) from the CPU can be inverted and used instead of WR. This requires, however, that write data be valid before CAS.

From Figure 4, it can be seen that the minimum high time for MREQ between opcode fetch and refresh cycles is 105ns for the Z80A. For systems that use MREQ to generate RAS, this is not sufficient to satisfy RAS precharge time requirements of the slower RAMs. However, as will be shown in the design example, relatively simple logic can be used to extend RAS high time between these cycles.
Figure 3  Z80-CPU op code fetch cycle timing at 2.5 MHz clock

Figure 4  Z80A-CPU op code fetch cycle timing at 4 MHz clock

NOTE: ALL TIMING IN ns  ASSUME RISE/FALL TIME: 15ns
Figure 5  Z80-CPU read cycle timing at 2.5 MHz clock

Figure 6  Z80A-CPU read cycle timing at 4 MHz clock

NOTE: ALL TIMING IN ns
ASSUME RISE/FALL TIME: 15ns
Figure 7  Z80-CPU write cycle timing at 2.5 MHz clock

Figure 8  Z80A-CPU write cycle timing at 4 MHz clock
MEMORY CYCLE SELECTION

Selection of an operating mode is controlled by a combination of CAS and WRITE while RAS is active. The available modes in most 4K and 16K RAMs are a read cycle, a write cycle, a read-write cycle, and a read-modify-write cycle. For some of the newer 4K devices and the 16K device, another type of cycle known as page mode allows for faster access time by keeping the same row address and strobing successive column addresses onto the chip.

The read-modify-write cycle can be accomplished in less time than a read cycle followed by a write cycle because the addresses do not change in between. It is, therefore, possible to generate the write strobe as soon as the data modification is complete. In other words, data is read from a cell, modified, and then rewritten in its modified form into the same cell. In contrast, a read-write cycle does not require data to be valid at the output before the write operation is started.

In a write cycle, if the WRITE input is brought low before CAS (early write), the data is strobed in by CAS. In a delayed write cycle, the WRITE line goes low after CAS and data is strobed in with WRITE.
DYNAMIC RAM MEMORY ORGANIZATION

Careful attention must be given to dynamic RAM memory array layout. Page decoding, power line routing and filtering, noise suppression and generation, buffer drive requirements, and system upgrading are all important considerations during the design phase.

If a memory array consisting of 4K devices exceeds 4K bytes (one page), it will be necessary to configure multiple rows. Each row, or page, is selected by decoding address lines A12-A15. If the system is intended to be upgraded with 16K devices, the chip select line (CS) should not be used for device selection. Instead, RAS should be gated to the selected 4K bank with CAS being applied to all devices. (Chips that receive CAS but no RAS will be unselected.) The CS line should be distributed to all devices and tied to ground. It can then be used as the seventh address line when upgrading to 16K RAMS.

The CAS line is used to control the output buffer in a configuration where the outputs are or-tied. If true data is still available from a previous cycle (assuming latched output 4K RAMS), then CAS deselects these devices if they are not being accessed during the current cycle. Note also that if RAS is inactive and CAS active, the only function that is performed is to change any true outputs to the high impedance state. Figure 9 shows the logic for one data bit in an 8K-byte system utilizing two banks of 4K RAM devices.

The absence of an output latch on most 16K RAMS can allow for simplification in system design. Unlike the latched 4K devices which need an extra cycle to clear the latch, the 16K non-latched device maintains data valid only during the time the CAS clock is active. Each memory cycle, therefore, can be maintained as an independent cycle, allowing the data input and output pin to be directly connected. This is assuming, however, that the write line goes true before CAS (early write mode).
Figure 9  Partial memory configuration in 8k byte system
All inputs on most dynamic RAMS are TTL compatible (on some 4K devices RAS, CAS, and the WRITE line require a 2.7 volt minimum logic 1 level which will require a pull-up resistor on the TTL driver). These TTL inputs, however, do not source current; but instead, present purely capacitive loads. This capacitance will vary between 5pf and 10pf on most 4K and 16K devices. With a large number of RAMS in a memory array, capacitive loading becomes a consideration. A 16K byte memory array made up of 4K devices will present from 150pf to 250pf of input capacitance to the input buffers. Most TTL outputs are not specified above 50pf. Therefore, a TTL driver must be used that can provide enough charging and discharging current to achieve the required voltage transition within the allotted time. A fairly accurate calculation can be made for determining the required drive current by using the standard relationship between the charging current $i$, the capacitance $C$, the voltage transition $V$, and the allotted time $T$:

$$i = \frac{\Delta V}{\Delta T}$$

For example, if the worst case capacitance on an address line is 250pf and it is required to change this address line within a 60ns period from zero volts to 3 volts, the driving current is:

$$i = \frac{250 \times 10^{-12}}{60 \times 10^{-9}} = 12\text{mA}$$

The power consumption of dynamic RAMS, which generally varies from 350mw to 1 watt, depends on the state of the RAS and CAS clocks. The device draws minimum current when these clocks are inactive (standby mode). At each transition of the clocks, the device will draw current. This current corresponds to the precharging of these lines which represent large capacitance loads. During standby, the power consumption is usually less than 20mw. Also, because of this very low power dissipation when the clocks are turned off, the technique of decoding RAS to selected chips results in a sizable decrease in power consumption (approximately 60% of all active power is due to RAS and only 40% is due to CAS). Because the memory is dynamic, the power dissipation is a function of the rate of memory access and, therefore, operating frequency.
The resulting current spike, which occurs when the RAS and CAS clocks go through their negative transitions, is coupled onto the power supply busses causing noise throughout the system. To compensate for this noise, high-frequency ceramic bypass capacitors should be placed within the memory array. A good practice is to supply a .1uf capacitor every other device between +12V and ground. Alternating between these capacitors, a Decoupling on the +5V line to prevent noise from affecting TTL logic should consist of a .01uf capacitor every 4 or 5 devices. For low frequency decoupling, a 10uf tantalum capacitor between +12 and ground should be supplied every 16 devices with a 10uf tantalum between -5V and ground every 32 devices.

The use of a multi-layer board with internal power and ground planes would be beneficial in a dynamic RAM system. However, proper routing of power lines on a two-sided card should provide satisfactory results. It has been found that bussing the +12 volt and ground lines both horizontally and vertically at every device will reduce noise and greatly improve RAM performance. The -5 and +5 volt lines need not be bussed in this fashion since they are less heavily loaded and are less likely to see current spikes.

Keeping the layout as small as possible and locating the address and data bus buffers as close to the array as possible will also reduce potential ringing and reflections.

Figure 10 represents a typical expandable RAM interface for a total memory capability of either 16K using 4K devices or 64K using 16K devices. The multiplexing of address lines is done by "wire-oring" 8T97 drivers and controlling the tri-state input for row to column switching. Since the minimum voltage on any RAM input is -1 volt, a small series resistor (about 30ohms) is inserted on each RAM address line to surpass any undershoot that might occur. When using 4K RAMs, the lower section of the 74S139 decoder selects the desired 16K quadrant by decoding address lines A14 and A15. The upper section of the decoder selects the desired 4K bank in this quadrant by decoding address lines A12 and A13. When using 16K RAMs, the lower section of the decoder is not used and the upper section decodes the desired 16K
Figure 10 Z80A-16K/64K dynamic ram interface
quadrant with address lines A14 and A15. The latch on the upper address line is used to prevent potential spikes on the RAS lines as MREQ and the address lines change at the end of the cycle.

The use of 8T97 drivers, with an external pull-up resistor, will insure proper logic level and capacitance drive capability. When using 4K devices, memory address line 6 (MA6) is not needed and tied to ground (this is the chip select line on 4K RAMs). When using 16K RAMs, this line is the 7th address line (A6 for row and A13 for column).
SLOW MEMORY INTERFACE

When working with memory devices with long access times (2708 EPROMS with a 450ns max access time, for example), it will be necessary to add wait states to Z80A timing. Figure 11 shows how a JK flip-flop can be configured for adding one wait state (250ns with a 4MHz clock) to each memory cycle. When using dynamic memories that have access times between 250 and 350ns, it is only necessary to add wait states for Op Code fetch cycles, since this cycle is the critical one in terms of memory access requirements. In this case, the logic in Figure 11 can be controlled by M1 instead of MREQ to accommodate these memories.
Figure 11 Adding one wait state to each memory cycle
DESIGN EXAMPLE

A typical design is presented to demonstrate a technique for dynamic RAM interface to the Z80A operating at 4MHz. Of the several approaches that could have been used for generating the timing signals needed for this interface, the tradeoffs for considering this approach consisted of the following:

1. Monostable-multivibrators could have been used to generate the time delays for the MUX switching and CAS signals, but one-shots are hard to adjust and are less reliable than other approaches.

2. The inherent delay in low power TTL gates could be used for this timing, but predictable timing intervals are hard to achieve at 4MHz.

3. A tapped delay line produces very accurate timing signals but is less attractive from a cost standpoint.

A synchronous technique has been chosen for this design because it generates accurate signals with a minimum of logic complexity and produces predictable results from system to system. The approach is to generate the CPU 4MHz clock from an 8MHz source. This 20 clock is then divided by two and used with the resulting 0 clock to generate the MUX switch and CAS signals after RAS has been generated from the fall of MREQ. Figure 12 is a schematic diagram of this interface. Figure 13 indicates the timing relationship involved.

The ROW Address Strobe (RAS) is generated at the fall of MREQ. On the next rising edge of the 0 clock, 'A' flip-flop is clocked to generate the signal used to switch the multiplexer from ROW addresses to Column addresses. The following falling edge of the 20 clock is used to generate the CAS signal (B flip-flop). Flip-flop C is used to insure sufficient RAS precharge time, which must be taken into account since MREQ has a minimum high time of 100ns between Op Code fetch and refresh cycles and MREQ, therefore, cannot be used to set RAS high. With CAS true, the trailing edge of RAS is clocked high with the 0 clock. This will extend the RAS high time to approximately 150ns.
Figure 12  Z80A dynamic ram interface
Figure 13  Z80A dynamic ram interface timing
This basic logic structure is configured into a microcomputer system and is seen in Figure 14. For simplicity, only the logic pertaining to the RAM interface is shown. Additional logic consisted of monitor software and a serial I/O interface to a CRT terminal.

Calculated timing parameters matched measured data quite accurately. Figures 15 through 20 indicate recordings taken during the Op Code fetch and refresh cycles at room temperature and at a Vcc of +5.0 volts.

From Figure 19, it can be seen that the interval between the leading edge of RAS and the leading edge of the switch MUX signals is approximately 50ns. The calculated interval is 35ns minimum and is consistent with the ROW address hold time tRAH (see Z6116 Product Specification) of all RAMs that are access time compatible with the Z80A.

At the leading edge of the switch MUX signal, the RAM addresses are switched from ROW to Column addresses. Assuming the column address set up time (tASC) to be zero (consistent with most dynamic RAMs), the interval for address switching is approximately 70ns as confirmed from calculated and measured data (see Figures 13 and 19). Scope triggering records both Row and Column addresses which appear to be superimposed on a typical RAM address line as seen in Figure 17.

Since the RAS to CAS interval exceeds the tRCD max value of most access-compatible RAMs, the RAM access time is measured from the leading edge of CAS. RAMS with CAS access times of 150ns or less should be compatible with this interface approach. If it is desired to keep the RAS to CAS interval within or closer to the tRCD max limit, a 40 clock could be applied to the clock input of Flip-Flop B (Figure 12) instead of the 20 clock. This would reduce the RAS to CAS interval to approximately 65ns.
Figure 13  Z80A 16K ram interface
The recordings also show the relation between MREQ and RAS high time between Op Code fetch and refresh cycles. The calculated value for RAS high time was 150ns while the measured value was approximately 170ns. This allows adequate RAS precharge time for all access-compatible RAMs.

A composite of all the major control signals, including one data line, is seen in Figure 20. This recording, done with a logic analyzer, shows the relative relation of these signals during the Op Code fetch and refresh cycles. Notice, that during refresh, only RAS is active with the switch MUX and CAS signals disabled.

CONCLUSIONS

The high density, reduced standby power, and reduced cost per bit of 16-pin dynamic RAMs have made these devices suitable for an increasing number of applications. Their attractiveness is also enhanced by the ease of upgrading from 4K to 16K devices. With this increased usage, however, the interface logic between the memory array and the microprocessor becomes an important consideration. The Z80A has simplified this interface. Internal logic operates totally transparent to CPU operation, supplying refresh capability without the need for a refresh counter and its associated multiplexer. This interface can be configured with just five standard TTL gates to obtain synchronous generation of the RAS, CAS, and the multiplexer switching signal. Additional design attention must be given to the RAM layout which can have a dramatic effect on system performance. Dynamic RAMs tend to be noise generators as well as being noise sensitive. However, with proper attention to filtering, power line routing, and array organization, dynamic RAM memory can provide a cost effective solution for high-density storage.
Although interrupts are not necessarily the fastest way to control I/O in a microcomputer system, they are often the most practical—even when several asynchronous external events must be serviced in preference to ongoing calculations.

Interrupt processing itself is, of course, a function of hardware architecture, but it must be supported by special routines in the user's software. To do that efficiently requires detailed knowledge of how an interrupt functions.

Say a peripheral generates an interrupt condition when a character becomes available on the Z80-SIO (serial-I/O) receiver. When the connected peripheral's interrupt-enable input line is high and its internal interrupt circuitry is enabled, it activates the interrupt line of the Z80 CPU.

The processor samples the interrupt line on the last T state of the last machine cycle in every instruction. If the interrupt line (INT) is active, the interrupt-enable flip-flop in the Z80 is set and the data-bus request line (BUSRQ) is inactive, the CPU acknowledges the interrupt by entering a special M1 cycle called the interrupt-acknowledge cycle. An I/O request is then made during the last T state of this cycle to the device, which is now able to put its vector on the data bus. This vector, together with the I register, forms a 16-bit pointer in the interrupt service routine's starting-address table (ISR-SAT). The Z80 CPU then obtains a 2-byte address from the table and jumps to that address.

But before the interrupt can be processed properly, the user has to prepare the ground:

1. An interrupt "page" must be chosen and the I-register programmed accordingly.
2. The device interrupt vector must be programmed.
3. An entry (or several) must be made in the ISR-SAT.

When the interrupt has been acknowledged, the machine state can be described as follows:

- The user program has been interrupted.
- Control has been passed to the proper interrupt service routine (to which the table entry for the

interrupting device is pointing).

- All maskable interrupts are disabled, as they would be after a Disable Interrupts instruction.
- The program then executes the interrupt handler routine. Since the Z80 CPU does not preserve the state of the system automatically when interrupts occur, the interrupt handler must do this job, in one of three ways.

First, registers and flags may be saved on the stack with the following sequence of instructions and T cycles:

```
PUSH AF 11
PUSH HL 11
PUSH DE 11
PUSH BC 11
PUSH IX 15
PUSH IY 15
Total T cycles: 74
```

When this method is used, a similar sequence of POP instructions restores the stack at the end of the service routine:

```
POP iY 14
POP IX 14
POP BC 10
POP DE 10
POP HL 10
POP AF 10
Total T cycles: 68
```

Second, other RAM locations may be used for the same purpose:

```
LD (ASAVE),A 13
LD (HLSAVE),HL 16
LD (DESAVE),DE 20
LD (BCSAVE),BC 20
LD (IXSAVE),IX 20
LD (IXSAVE),IY 20
Total T cycles: 109
```

1. A daisy-chain mechanism resolves priority conflicts between Z80 peripherals. A device can only interrupt when its lEI line is high. While being serviced, the interrupting device (highlighted) keeps its lEO line low, preventing lower-priority devices (gray) from issuing an interrupt.

The flat register F, however, cannot be saved directly in the RAM method—a serious limitation. Furthermore, a sequence taking 109 T states could be prohibitively long in some applications.

Third, when interrupt-overhead time and program space must both be minimized, the Z80 CPU's alternate register set may be used to save registers and flags. Here, EXX exchanges the contents of the BC, DE and HL registers with the contents of the BC', DE' and HL' registers, respectively, while EX AF, AF' exchanges the contents of AF and AF'.

This operation requires only eight T states, 2 bytes of code and no additional stack space. However, this method is of limited use when multiple interrupts are needed since only one interrupt handler may use the alternate register set; this excludes the use of nested interrupts. The method also assumes that the alternate register set is not already used by the program that is being interrupted.

Since only those registers whose content is destroyed by the interrupt service routine need to be preserved, a combination of the three methods is frequently used.

More interruptions

If successive interrupts are to be allowed, the interrupt handler must explicitly enable them with an Enable Interrupts (EI) instruction. This can be done anywhere in the program after the first interrupt occurs, and before the next one is expected.

In most cases, however, exactly where in the main program an interrupt occurs is not known. Consequently, it is good practice to enable interrupts in the interrupt handler itself. If the interrupt handler executes an EI instruction immediately, higher-priority devices could interrupt as early as the instruction immediately after EI, producing nested interrupts. Program-timing calculations must take this into account.

When several peripheral devices operate simultaneously in interrupt mode on a Z80 CPU, interrupts from any device may have to be acknowledged within a very short time to avoid data loss or other I/O malfunctions. In such cases, interrupts should be reenabled as early as possible in the interrupt handler to minimize the interval during which real-time events will not be recognized. This time will not exceed the value $T_{\text{im}}$, which is defined as follows:

$$T_{\text{im}} = \text{Interrupt-acknowledge time} + \text{instruction time for EI} + \text{instruction time for following instruction}$$

The $T_{\text{im}}$ can be minimized by forcing the instruction following EI to be a NOP or any other instruction of four T states.

When the programmer allows nested interrupts to occur, he must remember that only one service routine may use the alternate register set for preserving the interrupted program state. If all the interrupt handlers save registers on the stack, he must verify that
2. All interrupt routines start by saving program status, which they restore at the end. They terminate with Enable Interrupt (EI) and Return from Interrupt (RETI) instructions—what happens in between depends largely on the specific application.

3. Sectors on a floppy disk contain not only data (highlighted) but also pre and postambles, as well as control information.

4. A Z80A CTC chip generates interrupts for a disk controller. Its channel 0 counts index or sector pulses (pin 23), while the cascaded channels 1 and 2 provide a wide timing range.

Watch for the daisy chain

The interrupt handler must terminate with RETI, which is decoded by the interrupting device and allows the device to leave the interrupt state. In particular, it reactivates the lower daisy chain by resetting lines IEI (Interrupt Enable In) and IEO (Interrupt Enable Out) so that interrupts from lower-priority devices can be recognized (Fig. 1).

Keep in mind that EI and RETI do different things. The EI instruction does a general interrupt enable on the Z80 CPU for maskable interrupts. This does not imply, however, that all Z80 peripheral devices are allowed to interrupt—only those that have an active IEI line. In an interrupt-service routine, therefore, only devices higher in the chain than the interrupting device may, in turn, interrupt after EI and before RETI.

The RETI instruction is the only way to reestablish the interrupt daisy chain, below the interrupting device, by software. By resetting the interrupt-under-service flip-flop inside the peripheral, RETI affects the state of the interrupting device, while EI affects the state of the Z80 CPU.

Fig. 2 shows a generalized interrupt routine that summarizes the discussed steps. But the flow chart should not be regarded as a rigid model—interrupt handlers can differ widely from one application to another.

Making tracks—carefully

A hard-sectored floppy-disk driver provides many opportunities for interrupt-service routines: Sector count, head-load-delay time-out and track stepping in particular require interrupt techniques to achieve acceptable performance levels.

One problem stems from slight head misalignments, which cause the physical location of sector data to vary slightly with respect to the sector hole. The impact on data readability may be severe because flexible diskettes are likely to be read on several drives with different alignment characteristics.

For example, if data are written “early” with respect to the sector hole, the read gate may be turned on too late to catch the start bit, and synchronization will occur on the first ONE bit in the data field. A sector-address error or a CRC error are typical results. If data are written “late” with respect to the sector hole, the last bytes of the previous sector may still be under the read head when the read gate is turned on.

The best point to turn on the read gate is at the

the stack area is large enough to prevent overflow.

If nested interrupts are not desirable, the EI instruction is usually inserted at the end of the interrupt service routine, immediately before the Return from Interrupt (RETI) instruction. Any pending interrupt will then be delayed until the end of the interrupt handler, that is, until RETI has been executed.
Seventy-five microsecond delays; handles the sector hole.

The Interrupt routines generate the proper delay to look for the sector hole—not of the sector to be read, but the one before. Then the program adds 0.9 times the length of a sector and, at that moment, switches from an interrupt to a polling technique. This approach assures that the CPU can execute other tasks as long as possible, without running the risk of missing a sector hole.

It would have been simpler to program the Z80 CTC to interrupt on the requested sector hole rather than on the previous one. In that case, however, the interrupt overhead delay, added to the time required to preserve registers and flags and to set up the Z80 CTC time delay, would be in the order of 186 T states (Fig. 6).

If the controller must operate both on a 2.5-MHz (Z80) and a 4-MHz (Z80A) processor, the described method becomes unacceptable because the variable overhead time between the moment the sector pulse occurs and the moment the preamble-delay countdown starts. Instead, the sector hole is detected by software polling, which only introduces a CPU clock-dependent delay of 86 to 101 T states (Fig. 7). The polling loop, however, is entered about 4.5 ms after the previous hole has been encountered, to return control to the main program for as long as possible.

Polling and interrupt techniques can be combined in this case because sector holes are regularly spaced on a disk. Knowing when one hole flies by is, in theory, enough to predict the start of any following sector; in practice, however, speed variation, electrical delays

middle of the preamble (Fig. 3)—and this point must be found with good accuracy.

A Z80 CTC (counter-timer circuit) may be used to generate floppy-disk controller interrupts. In this application, three of its four programmable counters are needed.

Counter channel 1 is driven by a 1.4056-μs periodic signal (derived from a standard 11.3828-MHz crystal oscillator, divided by 16) that is programmed to count 22 (=16I) such pulses. The counter, therefore, reaches a full count every 32 μs, which corresponds to the length of 1 byte.

The second counter is driven in cascade by the first counter's output (pin 9) and programmed to generate an interrupt after a varying number of byte units on pin 12 (line INT).

The sector pulse line from the floppy drive is connected to the counter input of CTC channel 0 (pin 23) to generate an interrupt on any desired sector. The sector pulse line is also connected to a Z80 PIO bit on this same controller so that the presence of a sector hole may be detected by a simple port read followed by a bit test.

Several routines are used in the sector-read operation of a typical hard-sectoring floppy-disk driver. The program (Fig. 5) assumes that the number of holes between the current sector and the requested sector has been computed in a previous routine, the result being held in a variable named COUNT.

The interrupt routines generate the proper delay to look for the sector hole—not of the sector to be read, but the one before. Then the program adds 0.9 times the length of a sector and, at that moment, switches from an interrupt to a polling technique. This approach assures that the CPU can execute other tasks as long as possible, without running the risk of missing a sector hole.

It would have been simpler to program the Z80 CTC to interrupt on the requested sector hole rather than on the previous one. In that case, however, the interrupt overhead delay, added to the time required to preserve registers and flags and to set up the Z80 CTC time delay, would be in the order of 186 T states (Fig. 6).

If the controller must operate both on a 2.5-MHz (Z80) and a 4-MHz (Z80A) processor, the described method becomes unacceptable because the variable overhead time between the moment the sector pulse occurs and the moment the preamble-delay countdown starts. Instead, the sector hole is detected by software polling, which only introduces a CPU clock-dependent delay of 86 to 101 T states (Fig. 7). The polling loop, however, is entered about 4.5 ms after the previous hole has been encountered, to return control to the main program for as long as possible.

Polling and interrupt techniques can be combined in this case because sector holes are regularly spaced on a disk. Knowing when one hole flies by is, in theory, enough to predict the start of any following sector; in practice, however, speed variation, electrical delays
6. Timing analysis shows that detecting sectors purely by
interrupts requires 186 T states—too much for a CPU-
clock-independent floppy controller.

7. Detecting sectors by polling takes less time; the exact
number of T states depends on the timing of the sector
pulse with respect to program execution.

and mechanical delays introduce too many uncertain-
ties. Still, the approximate knowledge of the requested
hole's occurrence permits the calling program to
utilize additional CPU time before the polling loop for
sector-hole sensing takes over. This provides the
preamble delay-time count with an accurate starting
point.

The presence of one index hole on the disk, whose
position with respect to the first encountered hole is
not known, requires special attention. When the Z80
CTC interrupts, its internal counter is reloaded and
remains enabled, and any pulse on the Z80 CTC input
line will decrement the counter, even though the device
may not have left the interrupt state. To avoid
spurious interrupts from the index hole, the Z80 CTC
is always reset in all Z80 CTC interrupt-service
routines, even though this may not always be neces-
sary.

---

### How useful?

<table>
<thead>
<tr>
<th></th>
<th>Circle No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate design application</td>
<td>550</td>
</tr>
<tr>
<td>Within the next year</td>
<td>551</td>
</tr>
<tr>
<td>Not applicable</td>
<td>552</td>
</tr>
</tbody>
</table>

ELECTRONIC DESIGN 6, March 15, 1980 2-151
Introduction  The Zilog Z8000 CPU microprocessor is a major advance in microcomputer architecture. It offers many minicomputer and mainframe features for the first time in a microprocessor chip. This tutorial describes the Z8000 CPU with emphasis placed on those features that set it apart from its microprocessor predecessors. For a detailed description of all Z8000 CPU features, consult the Zilog publications listed in the bibliography at the end of this tutorial.

The features to be discussed are grouped into four areas: CPU organization, handling of interrupts and traps, use of memory, and new instructions and data capabilities.

Before discussing these features in more detail, a word about nomenclature is in order. The term Z8000 refers to the concept and architecture of a family of parts. Zilog has adopted the typical conductor industry 4-digit designation for Z8000 Family parts, while also keeping the traditional 3-letter acronym that proved so popular for the Z-80 Family. Thus, the 48-pin version of the Z8000 CPU is called the Z8001 CPU; the 40-pin version is known as the Z8002 CPU.

CPU Organization  The Z8000 CPU is organized around a general-purpose register file (Figure 1). The register file is a group of registers, any one of which can be used as an accumulator, index register, memory pointer, stack pointer, etc. The only exception is Register 0, as explained later.

Flexibility is the major advantage of a general-purpose register organization over an organization that dedicates particular registers to each function. Computation-oriented routines can use general registers as accumulators for intermediate results whereas data manipulation routines can use these registers for memory pointers.

Dedicated registers, however, have a disadvantage: when more registers of a given type are needed than are supplied by the machine, the performance degrades by the extra instructions to swap registers and memory locations. For example, a processor with two index registers suffers when three are needed because a temporary variable in memory (or in another register) must be used for the third index. When the third index is needed, it must be swapped into an index register. In contrast, on a general-register machine three of the registers could be dedicated for index use. In addition, since the need for index registers may vary over the course of a program, a general-register architecture, such as the Z8000, can be adapted to the changing needs of the computation with respect to the number of accumulators, memory pointers and index registers. Thus flexibility results in increased performance and ease of use.

In addition, the registers of the Z8000 are organized to process 8-bit bytes, 16-bit words, 32-bit long words and 64-bit quadruple words. This readily accommodates applications that process data of variable sizes as well as different tasks that require different data sizes.

Although all registers can—in general—be used for any purpose, certain instructions such as Subroutine Call and String Translation make use of specific registers in the general register file, and this must be taken into account when these instructions are used.
Figure 1. CPU Organization
The Z8000 CPU also contains a number of special-purpose registers in addition to the general-purpose ones. These include the Program Counter, Program Status registers and the Refresh Counter. These registers are accessible through software and provide some of the interesting features of Z8000 CPU architecture.

All general-purpose registers can be used as accumulators, and all but one as index registers or memory pointers. The one register that cannot be used as an index register is Register 0. Specifying Register 0 is used as an escape mechanism to change the address mode from IR to IM, from X to DA, or—with Load instructions—from BA to RA. This has been done so that the two addressing mode bits in the instruction can specify more than four addressing modes for the same opcode.

The Z8000 CPU register file can be addressed in several groupings: as sixteen byte registers (occupying the upper half of the file only), as sixteen word registers, as eight long-word registers, as four quadruple-word registers, or as a mixture of these. Instructions either explicitly or implicitly specify the type of register. Table 1 illustrates the correspondence between the 4-bit source and destination register fields in the instruction (Figure 4) and the location of the registers in the register file (Figures 2 and 3).
Register Organization (Continued)

Note that the byte register-addressing sequence (most significant bit distinguishes between the two bytes in a word register) is different from the memory addressing sequence (least significant bit distinguishes between the two bytes in a word). Long-word (32-bit) and quadruple-word (64-bit) registers are addressed by the binary number of their starting word registers (most significant word). For example, RR6 is addressed by a binary 6 and occupies word registers 6 and 7.

System/Normal Mode of Operation

The Z8000 CPU can run in one of two modes: System or Normal. In System Mode, all of the instructions can be executed and all of the CPU registers can be accessed. This mode is intended for use by programs that perform operating system type functions. In Normal Mode, some instructions, such as I/O instructions, are not all allowed, and the control registers of the CPU are inaccessible. In general, this mode of operation is intended for use by application programs. This separation of CPU resources promotes the integrity of the system since programs operating in Normal Mode cannot access those aspects of the CPU which deal with time-dependent or system interface events.

Normal Mode programs that have errors can always reproduce those errors for debugging purposes by simply re-executing the programs with their original data. Programs using facilities available only in System Mode may have errors due to timing considerations (e.g., based on the frequency of disk requests and disk arm position) that are harder to debug because these errors are not easily reproduced. Thus a preferred method of program development would be to partition the task into that portion which can be performed without recourse to resources accessible only in System Mode (which will usually be the bulk of the task) and that portion requiring System Mode resources. The classic example of this partitioning comes from current minicomputer and mainframe systems: the operating system runs in System Mode and the individual users write their programs to run in Normal Mode.

To further support the System/Normal Mode dichotomy, there are two copies of the stack pointer—one for the System Mode and another for Normal. Although the stacks are separated, it is possible to access the normal stack registers while in the System Mode by using the LDCTL instruction.

Status Lines

The Z8000 CPU outputs status information over its four status lines (ST4-ST7) and the System/Normal line (S/N). This information can be used to extend the addressing range or to protect accesses to certain portions of memory. The types of status information and their codes are listed in Table 2.

Status conditions are mutually exclusive and can, therefore, be encoded without penalty. Most status definitions are self-explanatory. One code is reserved for future enhancements of the Z8000 Family.

Extension of the addressing range is accomplished in a Z8000 system by allocating physical memory to specific usage (program vs. data space, for example) and using external circuitry to monitor the status lines and select the appropriate memory space for each address. For example, the direct addressing range of the Z8002 CPU is limited to 64K bytes; however, a system can be configured with 128K bytes if additional logic is used, say, to select the lower 64K bytes for program references and the upper 64K bytes for data references.

<table>
<thead>
<tr>
<th>ST3-ST0</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>Internal operation</td>
</tr>
<tr>
<td>0 0 1</td>
<td>Memory refresh</td>
</tr>
<tr>
<td>0 1 0</td>
<td>I/O reference</td>
</tr>
<tr>
<td>0 1 1</td>
<td>Special I/O reference</td>
</tr>
<tr>
<td>1 0 0</td>
<td>Segment trap acknowledge</td>
</tr>
<tr>
<td>1 0 1</td>
<td>Non-maskable interrupt acknowledge</td>
</tr>
<tr>
<td>1 1 0</td>
<td>Non-vectorized interrupt acknowledge</td>
</tr>
<tr>
<td>1 1 1</td>
<td>Vectored interrupt acknowledge</td>
</tr>
<tr>
<td>1 0 0</td>
<td>Data memory request</td>
</tr>
<tr>
<td>1 0 1</td>
<td>Stack memory request</td>
</tr>
<tr>
<td>1 0 1</td>
<td>Data memory request (EPU)</td>
</tr>
<tr>
<td>1 0 0</td>
<td>Stack memory request (EPU)</td>
</tr>
<tr>
<td>1 1 0</td>
<td>Instruction space access</td>
</tr>
<tr>
<td>1 1 0</td>
<td>Instruction fetch, first word</td>
</tr>
<tr>
<td>1 1 1</td>
<td>Extension processor transfer</td>
</tr>
<tr>
<td>1 1 1</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

Table 2
Protection of memory by access types is accomplished similarly. The memory is divided into blocks of locations and associated with each block is a set of legal status signals. For each access to the memory, the external circuit checks whether the CPU status is appropriate for the memory reference. The Z8010 Memory Management Unit is an example of an external memory-protection circuit, and it is discussed later in this tutorial.

The first word in an instruction fetch has its own dedicated status code, namely 1101. This allows the synchronization of external circuits to the CPU. During all subsequent fetch cycles within the same instruction (remember, the longest instruction requires a total of four word fetches), the status is changed from 1101 to 1100. Load Relative and Store Relative also have a status of 1100 with the data reference, so information can be moved from program space to data space.

The idea of incorporating the Refresh Counter in the CPU was pioneered by the Z-80 CPU, which performs a refresh access in a normally unused time slot after each opcode fetch. The Z8000 is more straightforward (each refresh has its own memory-access time slot of three clock cycles), and is more versatile (the refresh rate is programmable and capable of being disabled altogether).

The Refresh Register contains a 9-bit Row Counter, a 6-bit Rate Counter and an Enable Bit (Figure 5). The row section is output on AD0–AD8 during a refresh cycle. The Z8000 CPU uses word-organized memory, wherein A0 is only employed to distinguish between the lower and upper bytes within a word during reading or writing bytes. A0 therefore plays no role in refresh—it is always 0. The Row Counter is—at least conceptually—always incremented by two whenever the rate counter passes through zero. The Row Counter cycles through 256 addresses on lines AD1–AD8, which satisfies older and current 64- and 128-row addressing schemes, and can also be used with 256-row refresh schemes for 64K RAMs.

The Rate Counter determines the time between successive refreshes. It consists of a programmable 6-bit modulo-n prescaler (n = 1 to 64), driven at one-fourth the CPU clock rate. The refresh period can be programmed from 1 to 64 µs with a 4 MHz clock. A value of zero in the counter field indicates the maximum time between refreshes; a value of n indicates that refresh is to be performed every 4n clock cycles. Refresh can be disabled by programming the Refresh Enable Bit to be zero.

A memory refresh occurs as soon as possible after the indicated time has elapsed. Generally, this means after the T3 clock cycle of an instruction if an instruction execution has commenced. When the CPU does not have control of the bus (during the bus-request/bus-acknowledge sequence, for example), it cannot issue refresh commands. Instead, it has internal circuitry to record “missed” refreshes; when the CPU regains control of the bus it immediately issues the “missed” refresh cycles. The Z8001 and Z8002 CPU can record up to two “missed” refresh cycles.

Some instructions for which the overlap is logically impossible are the Jump instructions (because the following instruction location has not been determined until the instruction completes). Some instructions for which overlap is physically impossible are the Memory Load instructions (because the memory is busy with the current instruction and cannot service the fetch of the succeeding instruction).
Extended Instruction Facility

The Z8000 architecture has a mechanism for extending the basic instruction set through the use of external devices. Special opcodes have been set aside to implement this feature. When the CPU encounters instructions with these opcodes in its instruction stream, it will perform any indicated address calculation and data transfer, but otherwise treat the "extended instruction" as being executed by the external device. Fields have been set aside in these extended instructions which can be interpreted by external devices (called Extended Processing Units—EPUs) as opcodes. Thus by using appropriate EPUs, the instruction set of the Z8000 can be extended to include specialized instructions.

In general, an EPU is dedicated to performing complex and time consuming tasks in order to unburden the CPU. Typical tasks suitable for specialized EPUs include floating-point arithmetic, data base search and maintenance operations, network interfaces, graphics support operations—a complete list would include most areas of computing. EPUs are generally designed to perform their tasks on data resident in their internal registers. Moving information in and out of the EPU's internal registers, as well as instructing the EPU as to what operations are to be performed, is the responsibility of the CPU.

For the Z8000 CPU, control of the EPUs takes the following form. The Z8000 CPU fetches instructions, calculates the addresses of operands residing in memory, and controls the movement of data to and from memory. An EPU monitors this activity on the CPU's AD lines. If the instructions fetched by the CPU are extended instructions, all EPUs and the CPU latch the instruction (there may be several different EPUs controlled by one CPU). If the instruction is to be executed by a particular EPU, both the CPU and the indicated EPU will be involved in executing the instruction.

If the extended instruction indicates a transfer of data between the EPU's internal registers and the main memory, the CPU will calculate the memory address and generate the appropriate timing signals (AS, DS, MREQ, etc.), but the data transfer itself is between the memory and the EPU (over the AD lines). If a transfer of data between the CPU and EPU is indicated, the sender places the data on the AD lines and the receiver reads the AD lines during the next clock period.

If the extended instruction indicates an internal operation to be performed by the EPU, the EPU begins execution of that task and the CPU is free to continue on to the next instruction. Processing then proceeds simultaneously on both the CPU and the EPU until a second extended instruction is encountered that is destined for the same EPU (if more than one EPU is in the system, all can be operating simultaneously and independently). If an extended instruction specifies an EPU still executing a previous extended instruction, the EPU can suspend instruction fetching by the Z8000 CPU until it is ready to accept the next extended instruction: the mechanism for this is the STOP line, which suspends CPU activity during the instruction fetch cycle.

There are four types of extended instructions in the Z8000 CPU instruction repertoire: EPU internal operations; data transfers between memory and EPU; data transfers between EPU and CPU; and data transfer between EPU flag registers and CPU flag and control word. The last type is useful when the program must branch based on conditions determined by the EPU. Six opcodes are dedicated to extended instructions: 0E, 0F, 4E, 4F, 8E and 8F (in hexadecimal). The action taken by the CPU upon encountering these instructions is dependent upon an EPU control bit in the CPU's FCW. When this bit is set, it indicates that the system configuration includes EPUs; therefore, the instruction is executed. If this bit is clear, the CPU traps (extended instruction trap), so that a trap handler in software can emulate the desired operation.

In conclusion, the major features of this capability are, that multiple EPUs can be operating in parallel with the CPU, that the five main CPU addressing modes (Register, Immediate, Indirect Register, Direct Address, Indexed) are available in accessing data for the EPU; that each EPU can have more than 256 different instructions; and that data types manipulated by extended instructions can be up to 16 words long.
**Program Status Information**

The Program Status Information consists of the Flag And Control Word (FCW) and the Program Counter (PC). The Z8000 CPU uses one byte in FCW to store flags and another byte to store control bits.

**Arithmetic Flags.** Flags occupy the low byte in the FCW and are loaded, read, set and reset by the special instruction LDCTLB, RESFLG and SETFLG. The flags are:

- **C** Carry
- **Z** Zero
- **S** Sign (1 = negative; two’s complement notation is used for all arithmetic on data elements)
- **P/V** Even Parity or Overflow (the same bit is shared)
- **D** Decimal Adjust (differentiates between addition and subtraction)
- **H** Half Carry (from the low-order nibble)

**Interrupt and Trap Structure**

The Z8000 provides a powerful interrupt and trap structure. Interrupts are external asynchronous events requiring CPU attention, and are generally triggered by peripherals needing service. Traps are synchronous events resulting from the execution of certain instructions. Both are processed in a similar manner by the CPU.

The CPU supports three types of interrupts (non-maskable, vectored and non-vectored), three internal traps (system call, unimplemented instruction, privileged instruction) and a segmentation trap. The vectored and non-vectored interrupts are maskable.

The descending order of priority for traps and interrupts is: internal traps, non-maskable interrupts, segmentation trap, vectored interrupts and non-vectored interrupts.

**Effects of Interrupts on Program Status**

The Flag and Control Word and the Program Counter are collectively called the Program Status Information—a useful grouping because both the FCW and PC are affected by interrupts and traps. When an interrupt or trap occurs, the CPU automatically switches to the System Mode and saves the Program Status plus an identifier word on the system stack. The identifier supplies the reason for the interrupt. (The Z8002 pushes three words on the stack; the Z8001 pushes four words.)

After the pre-interrupt or “old” Program Status has been stored, the “new” Program Status is automatically loaded into the FCW and PC. This new Program Status Information is obtained from a specified location in memory, called the Program Status Area.

The Z8000 CPU allows the location of the Program Status Area anywhere in the addressable memory space, although it must be aligned to a 256-byte boundary. Because the Status Line code is 1100 (program reference) when the new Program Status is loaded, the Program Status must be located in program memory space if the memory uses this attribute (for example, when using the Z8010 Memory Management Unit or when separate memory modules are used for program and for data).

**Control Bits.** The control bits occupy the upper byte in the FCW. They are loaded and read by the LDCTL instruction, which is privileged in that it can be executed only in the System Mode. The control bits are:

- **NVIE** Non-Vectored Interrupt Enable
- **VIE** Vectored Interrupt Enable
- **S/N** System or Normal Mode
- **SEG** Segmented Mode Enable (Z8001 only)

The SEG bit is always 0 in the Z8002 even if the programmer attempts to set it. In the Z8001, a 1 in this bit indicates segmented operation. A 0 in the Z8001 SEG bit forces non-segmented operation and the CPU interprets all code as non-segmented. Thus, the Z8001 can execute modules of user code developed for the non-segmented Z8002.

<table>
<thead>
<tr>
<th>Location (In Bytes)</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-3</td>
<td>Not used (reserved for future use)</td>
</tr>
<tr>
<td>4-7</td>
<td>Unimplemented instruction has been fetched, causing a trap</td>
</tr>
<tr>
<td>8-11</td>
<td>Privileged instruction has been fetched in Normal Mode, causing a trap</td>
</tr>
<tr>
<td>12-15</td>
<td>System Call instruction</td>
</tr>
<tr>
<td>16-19</td>
<td>Not used</td>
</tr>
<tr>
<td>20-23</td>
<td>Non-maskable interrupt</td>
</tr>
<tr>
<td>24-27</td>
<td>Non-vectored interrupt</td>
</tr>
</tbody>
</table>

---

3-9
Effects of Interrupts on Program Status
(Continued)

Bytes 28-29 contain the FCW that is common to all vectored interrupts. Subsequent locations
contain the vector jump table (new PC for vectored interrupts). These locations are
directed in the following way: the 8-bit vector that the interrupting device has put on the lower byte of the Address/Data bus (AD₀-AD₇) is doubled and added to
PSAP + 30. Thus,

Vector 0 addresses PSAP + 30,
Vector 1 addresses PSAP + 32, and
Vector 255 addresses PSAP + 540.

In the segmented Z8001, the first 28 words of the Program Status Area (56 bytes) contain the Program Status Information (reserved word, FCW, segment number, offset), for the following interrupt conditions:

<table>
<thead>
<tr>
<th>Location (In bytes)</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-7</td>
<td>Not used (reserved for future use)</td>
</tr>
<tr>
<td>8-15</td>
<td>Unimplemented instruction has been fetched causing a trap</td>
</tr>
<tr>
<td>16-23</td>
<td>Privileged instruction has been fetched in Normal Mode causing a trap</td>
</tr>
<tr>
<td>24-31</td>
<td>System Call instruction</td>
</tr>
<tr>
<td>32-39</td>
<td>Segmentation trap (memory violation detected by the Z8010 Memory Management Unit)</td>
</tr>
<tr>
<td>40-47</td>
<td>Non-maskable interrupt</td>
</tr>
<tr>
<td>48-55</td>
<td>Non-vectored interrupt</td>
</tr>
</tbody>
</table>

Bytes 56-59 contain the reserved word and FCW common to all vectored interrupts. Subsequent locations contain the vector jump table (the new segment number and offset for all vectored interrupts). These locations are directed in the following way: the 8-bit vector that the interrupting device has put on the lower byte of the Address/Data bus (AD₀-AD₇) is doubled and added to PSAP + 60. Thus,

Vector 0 addresses PSAP + 60,
Vector 2 addresses PSAP + 64, and
Vector 254 addresses PSAP + 568.

Care must be exercised in allocating vector locations to interrupting devices; always use even vectors. Thus there are effectively only 128 entries in the vector jump table. (Figure 6 illustrates the Program Status Area.)

---

*Figure 6. Program Status Area*
The way a processor addresses and manages its memory is an important aspect in both the evaluation of the processor and the design of a computer system that uses the processor. Z8000 architecture provides a consistent memory address notation in combining bytes into words and words into long words. All three data types are supported for operands in the Z8000 instruction set. I/O data can be either byte- or word-oriented.

The Z8001 CPU provides a segmented addressing space with 23-bit addressing. The Z8010 Memory Management Unit can increase the address range of this processor. To support a memory management system, the Z8001 processor generates Processor Status Information.

In the Z8000 CPU, memory and I/O addresses are always byte addresses. Words or long words are addressed by the address of their most significant byte (Figure 7). Words always start on even addresses (A0 = 0), so both bytes of a word can be accessed simultaneously. Long words also start on even addresses.

Within a word, the upper (or more significant) byte is addressed by the lower (and always even) address. Similarly, within a long word, the upper (more significant) word is addressed by the lower address. Note that this format differs from the PDP-11 but is identical to the IBM convention.

There is good reason for choosing this format. Because the Z8000 CPU can operate on 32-bit long words and also on byte and word strings, it is important to maintain a continuity of order when words are concatenated into long words and strings. Making ascending addresses proceed from the highest byte of the first word to the lowest byte of the last word maintains this continuity, and allows comparing and sorting of byte and word strings.

Bit labeling within a byte does not follow this order. The least significant bit in a byte, word or long word is called Bit 0 and occurs in the byte with the highest memory address. This is consistent with the convention where bit n corresponds to position 2n in the conventional binary notation. This ordering of bit numbers is also followed in the registers.

These signals are also generated by the Z8002 CPU and—as mentioned earlier—can be used to increase the address range of this processor beyond its nominal 64K byte limit. It is not necessary to use a Z8010 Memory Management Unit with a Z8001. The segment number (upper six bits of the address) can be used directly by the memory system as part of the absolute address.

These issues are discussed in more detail in the following sections, along with a description of the method used to encode certain segmented addresses into one word. A brief comment on the use of 16K Dynamic RAMs with the Z8001 concludes this group of sections that deal with Z8000 CPU memory features.

In the Z8000 CPU, memory and I/O addresses are always byte addresses. Words or long words are addressed by the address of their most significant byte (Figure 7). Words always start on even addresses (A0 = 0), so both bytes of a word can be accessed simultaneously. Long words also start on even addresses.

Within a word, the upper (or more significant) byte is addressed by the lower (and always even) address. Similarly, within a long word, the upper (more significant) word is addressed by the lower address. Note that this format differs from the PDP-11 but is identical to the IBM convention.

There is good reason for choosing this format. Because the Z8000 CPU can operate on 32-bit long words and also on byte and word strings, it is important to maintain a continuity of order when words are concatenated into long words and strings. Making ascending addresses proceed from the highest byte of the first word to the lowest byte of the last word maintains this continuity, and allows comparing and sorting of byte and word strings.

Bit labeling within a byte does not follow this order. The least significant bit in a byte, word or long word is called Bit 0 and occurs in the byte with the highest memory address. This is consistent with the convention where bit n corresponds to position 2n in the conventional binary notation. This ordering of bit numbers is also followed in the registers.

These signals are also generated by the Z8002 CPU and—as mentioned earlier—can be used to increase the address range of this processor beyond its nominal 64K byte limit. It is not necessary to use a Z8010 Memory Management Unit with a Z8001. The segment number (upper six bits of the address) can be used directly by the memory system as part of the absolute address.

These issues are discussed in more detail in the following sections, along with a description of the method used to encode certain segmented addresses into one word. A brief comment on the use of 16K Dynamic RAMs with the Z8001 concludes this group of sections that deal with Z8000 CPU memory features.
Like most 16-bit microprocessors, the Z8000 CPU uses a 16-bit parallel data bus between the CPU and memory or I/O. The CPU is capable of reading or writing a 16-bit word with every access. Words are always addressed with even addresses ($A_0 = 0$). All instructions are words or multiple words.

The Z8000 CPU can, however, also read and write 8-bit bytes, so memory and I/O addresses are always expressed in bytes. The Byte/Word (B/W) output indicates whether a byte or word is addressed (High = byte). $A_0$ distinguishes between the upper and lower byte in memory or I/O. The most significant byte of the word is addressed when $A_0$ is Low (Figure 8).

For word operations in both the read and write modes, $B/W = \text{Low}$, $A_0$ is simply ignored and $A_1$-$A_{15}$ address the memory or I/O. For byte operations in the read mode, $B/W = \text{High}$, $A_0$ is again ignored, and a whole word (both bytes) is read, but the CPU internally selects the appropriate byte. For byte operations in the write mode, the CPU outputs identical information on both the Low ($AD_7-AD_0$) and the High ($AD_{15}-AD_8$) bytes of the Address/Data bus. External TTL logic must be used to enable writing in one memory byte and disable writing in the other byte, as defined by $A_0$. The replication of byte information for writes is for the current implementation and may change for subsequent Z8000 CPUs; therefore system designs should not depend upon this feature.

![Figure 8. Byte/Word Selection](image)

In organizing memory, segmentation is a powerful and useful technique because it forms a natural way of dividing an address space into different functional areas. A program typically partitions its available memory into disjointed areas for particular uses. Examples of this are storing the procedure instructions, holding its global variables, or serving as a buffer area for processing large, disk-resident data bases. The requirements for these different areas may differ, and the areas themselves may be needed only part of the time.

Segmentation reflects this use of memory by allowing a user to employ a different segment for each different area. A memory management system can then be employed to provide system support, such as swapping segments from disk to primary memory as requested (as in overlays), or in monitoring memory accesses and allowing only certain types of accesses to a particular segment. Thus, dealing with segments is a convenient way of specifying portions of a large address space.

When segmentation is combined with an address translation mechanism to provide relocation capability, the advantages of segmentation are enhanced. Now segments can be of variable user-specifiable sizes and located anywhere in memory.

The Z8001 generates 23-bit logical addresses, consisting of a 7-bit segment number and a 16-bit offset. Thus each of its six memory address spaces consists of 128 segments, and each segment can be up to 64K bytes. Different routines of a program can reside in different segments, and different data sets can reside in different segments. The Z8010 Memory Management Unit translates these logical addresses into physical-memory locations.
Long Offset and Short Offset Addressing

When a segmented address is stored in memory or in a register, it occupies two 16-bit words as previously described for the PC and PSAP. This is a consequence of the large addressing range. When a segmented address is part of an instruction in the Direct Address and Indexed Address Modes, there are two representations: Long and Short Offset addressing.

In the general unrestricted case of Long Offset, the segmented address occupies two words, as described before. The most significant bit in the segment word is a 1 in this case.

The Short Offset Mode squeezes the segment number and offset into one word, saving program size and execution time. Since 23 bits obviously don’t fit into a 16-bit word, the 8 most significant bits of the offset are omitted and implied to be zero. The most significant bit of the address word is made 0 to indicate Short Offset Mode. Short Offset addresses are thus limited to the first 256 bytes at the beginning of each segment. This may appear to be a severe restriction, but it is very useful, especially in the Index Mode, where the index register can always supply the full 16-bit range of the offset. Short Offset saves one instruction word and speeds up execution by two clock cycles in Direct Address Mode and three clock cycles in Indexed Mode.

Using the Z8010 Memory Management Unit

The Z8001 CPU can be combined with another 48-pin LSI device—the Z8010 MMU—for sophisticated memory management. The MMU provides address translation from the logical addresses generated by the Z8001 CPU to the physical addresses used by the memory. An address translation table, containing starting addresses and size information for each of the 64 segments, is stored in the MMU. The translation table can be written and read by the CPU using Special I/O instructions. The MMU thus provides address relocation under software control, making software addresses (i.e., logical addresses) independent of the physical memory addresses.

But the MMU provides much more than address relocation; it also monitors and protects memory access. The MMU provides a Trap input to the CPU and—if necessary—an inhibit signal (SUP) to the memory write logic when specific memory-access violations occur. The MMU provides the following types of memory protection:

- Accesses outside the segment’s allotted memory can be prevented.
- Any segment can be declared invalid or non-accessible to the CPU.
- Segments can be declared Read Only.
- By designating a segment as System Only, access can be prohibited during the Normal Mode.
- Declaring a segment Execute Only means it can be accessed only during instruction access cycles. Data or stack use is prohibited.
- Any segment can be excluded from DMA access.
- Segments can have a Direction And Write Warning attribute, which generates a trap when a write access is made in the last 256 bytes of its size. This mechanism can be used to prevent stack overflow.

Multiple MMUs must be used when more than 64 segments are needed. Thus, to support the full complement of 128 segment numbers provided for each Z8001 CPU address space, two MMUs are required. The MMU has been designed for multiple-chip configurations, both to support 128-segment translation tables and to support multiple translation table systems.

Note that the memory management features do not interfere with the ability to directly address the entire memory space. Once programmed, the MMU (or MMUs) translates and monitors any memory address generated by the CPU.

The MMU contains status bits that describe the history of each segment. One bit for each segment indicates whether the segment has been accessed; another bit indicates whether the segment has been written. This is important for certain memory management schemes. For example, the MMU indicates which segments have been updated and, therefore, must be saved on disc before the memory can be used by another program.

When translating logical addresses to physical memory addresses, the MMU must do the following: access its internal 64 x 32-bit RAM, using the segment number as the address, then add the 16 bits of RAM output to the most significant address byte (AD8–AD15) and finally place the result on its Address outputs. The least significant byte (AD0–AD7) bypasses the MMU.

The internal RAM access time is approximately 150 ns. Throughput delay is avoided by making the segment number available early: SN0–SN7 are output one clock period earlier than the address information on AD0–AD7.

In summary, the Z8000 CPU supports sophisticated memory management through such architectural features as the Status Lines, the R/W and S/N lines, Segment Trap input line, and early output of segment numbers.
**Using 16K Dynamic RAMs with the Z8001**  
Z8000 systems usually implement most of their memory with 16K x 1-bit dynamic RAMs that have time-multiplexed addresses (Zilog also manufactures this device—the Z6116). In Z8001-based systems with MMUs, CPU Address/Data lines AD1-AD7 supply row addresses, MMU address outputs A8-A14 supply column addresses, and MMU outputs A15-A23 are decoded to generate Chip Select signals that gate either RAS or CAS or both. Gating RAS reduces power consumption because all non-selected memories remain in the standby mode. But this technique requires that RAS must wait for the availability of the most significant address bits from the MMU. During refresh, the RAS decoder must be changed to activate all memories simultaneously.

Gating CAS does not achieve lower power consumption; however, this technique allows the use of slower memories because RAS can be activated as soon as the CPU address outputs are stable, without waiting for the MMU delay. Also, there is no need to change the CAS decoder during refresh.

---

**Data Types and Instructions**  
The Z8000 architecture directly supports bits, digits, bytes, and 16- or 32-bit integers as primitive operands in its instruction set. In addition, the rich set of addressing modes supports higher-level data constructs such as arrays, lists and records. The Z8000 also introduces a number of powerful instructions that extend the capabilities of microprocessors. The remaining sections of this paper describe Z8000 data types, addressing modes, and a selection of novel instructions.

**Data Types**  
Operands are 1, 4, 8, 16, 32, or 64 bits, as specified by the instruction. In addition, strings of 8- or 16-bit data can be manipulated by single instructions. Of particular interest are the increased precisions of the arithmetic instructions. Add and Subtract instructions can operate on 8-, 16-, or 32-bit operands: Multiply instructions can operate on 16- or 32-bit multiplicands; and Divide instructions can operate on 32- or 64-bit dividends. The Shift instructions can operate on 8-, 16-, and 32-bit registers.

**Addressing Modes**  
The rich variety of addressing modes offered by Z8000 architecture includes: Register, Immediate, Indirect Register, Direct Address, Index, Relative Address, Base Address, and Base Index. Three are of particular interest with respect to high-level data structures: Indirect Register, Base Address, and Base Index. These modes can be used for lists, records, and arrays, respectively.

**Indirect Register.** In this addressing mode, the contents of the register are used as a memory address. This mode is needed whenever special address arithmetic must be performed to reference data. Essentially, the address is calculated in a register and then used to fetch the data. For example, this mode is useful when manipulating a linked list, where each entry contains a memory pointer to the memory location of the next entry. Essentially, the pointer is loaded into a register and used to access the next item on the list. When the list item is large or has a complex structure, the Base Address or Base Index Modes can be used to access various components of the item.

**Base Address.** In this addressing mode, the memory address contained in the register (the base) is modified by a displacement in the instruction (known at compile time). This mode is useful, for example, in accessing fields within a record whose format is fixed at compile time.

**Base Index.** The memory address in this addressing mode is contained in a register (the base) and is modified by the contents of another register (the index). This mode can be useful in accessing the components of an array, because the index of the component is usually calculated during execution time—as a function of the index of a DO-Loop, for example.

**Index vs. Base Address.** In the Z8002 and in the Z8001 running non-segmented, these two addressing modes are functionally equivalent, because the base address and displacement are both 16-bit values.

When the Z8001 runs segmented, there is a difference: in the Index mode, the base address (including the segment number) is contained in the instruction, in either Short Offset or Long Offset notation. The 16-bit displacement stored in a register is then added to the offset in the base address to calculate the effective address. In the Base Address Mode, on the other hand, the 16-bit displacement is specified in the instruction and is added to the offset of the base address that is stored in a long-word register.
The Instruction Set

The Z8000 offers an abundant instruction set that represents a major advance over its predecessors. The Load and Exchange instructions have been expanded to support operating system functions and conversion of existing microprocessor programs. The usual Arithmetic instructions can now deal with higher-precision operands, and hardware Multiply and Divide instructions have been added. The Bit Manipulation instructions can access a calculated bit position within a byte or word, as well as specify the position statically in the instruction.

The Rotate and Shift instructions are considerably more flexible than those in previous microprocessors. The String instructions are useful in translating between different character codes. Special I/O instructions are included to manage peripheral devices, such as the Memory Management Unit, that do not respond to regular I/O commands. Multiple-processor configurations are supported by special instructions.

The following instructions exemplify the innovative nature of the Z8000 instruction set. A complete list of Z8000 instructions can be found in the reference materials listed at the end of this tutorial.

Load and Exchange Instructions.
Exchange Byte (EX) is practical for converting Z-80, 8080, 6800 and other microprocessor programs into Z8000 code, because the Z8000 uses the opposite assignment of odd/even addresses in 16-bit words.
Load Multiple (LDM) saves n registers and is useful for switching tasks.
Load Relative (LDR) loads fixed values from program space into data space.

Arithmetic Instructions.
Add With Carry and Subtract With Carry (ADC, SBC) are conventionally used in 8-bit microprocessors for multiprecision arithmetic operations. These instructions are rarely used with the Z8000 CPU because it has 16- and 32-bit arithmetic instructions.

Decrement By N and Increment By N (DEC, INC) are intended for address and pointer manipulation, but can also be used for Quick Add/Subtract Immediate with 4-bit nibbles. The flag setting is different from Add/Subtract instructions—as is conventional—in that the Carry and Decimal adjust flags are unaffected by the Increment and Decrement instructions to support multiple precision arithmetic.

Decimal Adjust (DAB) automatically generates the proper 2-digit BCD result after a byte Add or Subtract operation, and eliminates the need for special decimal arithmetic instructions.

Multiply (MULT) provides signed (two's complement) multiplication of two words, generating a long-word result; or of two long-words generating a quadruple word result. No byte multiply exists because it is rarely used and, after sign extension, can be performed by a word multiply.

Divide (DIV) provides signed (two's complement) division of a long word by another word, generating a word quotient and a remainder word; or of one quadruple-word by a long-word, generating a long-word quotient and long-word remainder.

Both Multiply and Divide use a conforming register assignment. That is, a multiply followed by a divide on the same registers is essentially a no-op. The register designation used in the operation description must be even for word operations and must be a multiple of four for long-word operations.

Logical Instructions.
Test Condition Code (TCC) performs the same test as a Jump instruction, but affects the least significant bit of a specified register instead of changing the PC.

Program Control Instructions.
Call Relative (CALR) is a shorter, faster version of Call, but with a limited range.
Decrement And Jump If Non-Zero (DJNZ) is a one-word basic looping instruction.
Jump Relative (JR) is a shorter, faster version of Jump, but with a limited range.

Bit Manipulation Instructions.
Test Bit, Reset Bit, Set Bit (BIT, RES, SET) are available in two forms: static and dynamic. For the static form, any bit (the position is defined in the immediate word of the instruction) located in any byte or word in any register or in memory can be set, reset or tested (inverted and routed into the Z flag).

For the dynamic form, any bit (the position is defined by the content of a register that is, in turn, specified in the instruction) located in any byte or word in any register, but not in memory, can be set, reset or tested.

Test And Set (TSET) is a read/modify/write instruction normally used to create operating system locks. The most significant bit of a byte or word in a register or in memory is routed into the S flag bit and the whole byte or word is then set to all 1s. During this instruction, the processor does not relinquish the bus.

Test Multi-Micro Bit and Multi-Micro Request/Reset (MBIT, MREQ, MSET, MRES) are used to synchronize the access by multiple microprocessors to a shared resource.
The Instruction Set (Continued)

such as a common memory, bus, or I/O device.

Note that the instruction MREQ (Multimicroprocessor Request) has nothing whatsoever in common with the MREQ (Memory Request) output from the Z8000 CPU.

Rotate and Shift Instructions.
The Z8000 CPU has a complete set of shift instructions that shift any combination of bytes or words, right or left, arithmetically or logically, by any meaningful number of positions as specified either in the instruction (static) or in a register (dynamic).

The CPU also has a smaller repertoire of rotate instructions that rotates bytes or words, either right or left, through carry or not, and by one bit or by two bits.

The instructions Rotate Digit Left and Rotate Digit Right (RLDB, RRDB) rotate 4-bit BCD digits right or left, and are used in BCD arithmetic operations.

Block Transfer and String Manipulation Instructions.

Translate And Decrement/Increment (TRDB, TRIB) is used for code conversion, such as ASCII to EBCDIC. These instructions translate a byte string in memory by substituting one string by its table-lookup equivalent. TRDB and TRIB execute one operation and decrement the contents of the length register; thus they are useful as part of loop performing several actions on each character.

Translate, Decrement/Increment and Repeat (TRDRB, TRIRB) are the same as TRDB and TRIB, except they repeat automatically until the contents of the length register become zero. They are therefore useful in straightforward translation applications.

Translate And Test, Decrement/Increment (TRTDB, TRTIB) tests a character according to the contents of the translation table.

Translate And Test, Decrement/Increment And Repeat (TRTDRB, TRTIRB) scans a string of characters. The first character is tested and, depending on the contents of the translation table, the process stops or skips to the next character. Stopped characters can be used for further processing.

I/O and Special I/O Instructions.
The Z8000 CPU has two complete sets of I/O instructions: Standard I/O and Special I/O. The only difference is the status information on the ST0-ST3 outputs. Standard I/O instructions are used to communicate with Z-Bus compatible peripherals. Special I/O instructions are typically used for communicating with the Memory Management Unit.

Both types of instructions transfer 8 or 16 bits and use a type of 16-bit addressing analogous to the Z8002 memory-addressing scheme: For word operations, A0 is always zero; in byte-input operations, A0 is used internally by the CPU to select the appropriate byte; in byte-output operations, the byte is duplicated in the high and low bytes of the address/data bus, and external logic uses A0 to enable the appropriate output device.

Bibliography

Selected Publications on the Z8000 Family
Z8001/Z8002 CPU Product Specification (00-2045)
Z8000 CPU Instruction Set (03-8020-01)
Z8000 PLZ/ASM Assembly Language Programming Manual (03-3055-01)
Z8010 Z-MMU Product Specification (00-2046)
Introduction

This application note describes the hardware design implementation of a small computer using the Zilog Z8002 16-bit microprocessor, ROMs/EPRoms and dynamic RAMs plus parallel and serial I/O devices. The interface requirements of the Z8002 to memory and to Z80A peripherals are described and design alternatives are given whenever possible. This design is similar in structure and is software compatible with the Zilog Z8000 Development Module (part number 05-6101-01).

The design uses a minimal number of TTL support devices and, whenever possible, gate functions have been combined into MSI circuits. The result is a design that uses MSI TTL circuits in a very efficient—but sometimes non-obvious—way that minimizes the package count. Because some of the design techniques may not be self-explanatory, an effort has been made to explain them.

General Structure

Figure 1 shows a block diagram of the design. The Z8002 16-bit microprocessor is the heart of the system. This high-performance CPU offers a regular architecture, a powerful instruction set, a sophisticated interrupt structure, and high throughput at a modest 4 MHz clock rate.


Fixed program and data information is stored in an array of 2K x 8 ROMs or EPRoms; 16 16K x 1 dynamic RAMs provide 32K bytes of read/write storage. Input/output is handled by five I/O devices. Two Z80A PIOs provide 4 byte-wide bidirectional ports (32 lines) with handshake control. A Z80A SIO provides two fully independent full-duplex asynchronous or synchronous serial data communications channels. Four counter/timers in the Z80A CTC relieve the processor from simple counting and timing tasks and generate the programmable baud-rates for the serial I/O channels. Eight switches can be interrogated and interpreted by the program.

The block diagram also indicates the various support functions. A crystal-controlled clock circuit generates a Z8002 and Z80A compatible clock signal plus two complementary TTL clocks. Address buffers drive the memory and I/O devices; address latches demultiplex the time-shared Address/Data bus.

The ROM array uses a One-of-Eight Address Decoder and the RAMs are driven by an address multiplexer and a RAS/CAS generator. The timing for all these functions originates in the bus control and timing circuit. The I/O devices are selected by an I/O decoder and receive Z80A equivalent control signals generated by the Z8002 to Z80A Control Translator. The following sections contain detailed descriptions of these circuits.
Figure 1. Block Diagram
Clock Generation

The Z8002 requires a continuously running clock with a frequency between 500 kHz and 4 MHz. Most Z8002 applications are performance oriented and the clock rate is therefore usually set close to the maximum limit of 4 MHz. At this frequency, the specified requirements for clock width (minimum of 105 ns High or Low) and clock transition times (maximum of 20 ns rise or fall) require careful attention. At 4 MHz, a 50% clock duty cycle is indirectly implied by this specification and the safest way to insure it is to start with a crystal oscillator frequency that is twice the clock rate, and divide it with a toggling flip-flop.

The Z8002 clock input is not TTL compatible. It requires a High level within 400 mV of $V_{CC}$. A resistive pull-up can achieve this level, but cannot guarantee the required rise-time (20 ns from 0.8 to 4.0 V) when driving the ≈30 pF clock input capacitance. The stringent rise time requirements dictate the use of an active pull-up as shown in Figure 2.

![Figure 2. Clock Generation](image-url)
CPU Output Buffering

The Z8002 outputs can sink 2 mA while maintaining TTL noise margins and can thus drive five LS-TTL inputs. All output delays are specified for a capacitive load of up to 50 pF. They increase by approximately 0.1 ns/pF of additional capacitive load.

Bidirectional buffering of the A/D lines. The Address/Data lines require Bus Transceivers, such as the LS243 Quad Non-Inverting Bus Transceiver with separate Enable inputs for the two directions (one active High; the other active Low), or the LS245 Octal Non-Inverting Bus Transceiver with a Direction Control input and an active Low Enable input.

Figure 3 shows the logic that controls four LS243 Quad Transceivers; Figure 4 shows the even simpler logic that controls two LS245 Octal Transceivers.

The bus transceivers are controlled by three CPU control outputs as shown in the following truth table.

<table>
<thead>
<tr>
<th>BUSACK</th>
<th>R/W</th>
<th>DS</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>H</td>
<td>L</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>H</td>
<td>L</td>
<td>H</td>
</tr>
<tr>
<td>L</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Unidirectional Buffering of CPU Control Outputs. The following CPU control outputs may require unidirectional buffering: AS, DS, MREQ, R/W, N/S, B/W.

The buffered signals must be 3-stated when BUSACK is Low. One LS365A or LS367A Hex 3-State Buffer can perform this function as shown in Figure 5. The LS244 Octal 3-State Buffer buffers eight signals, but uses a 20-pin package.

In a simple system, such as the one described here, BUSREQ is not used, so BUSACK is therefore always High. In a more complex system with direct memory access, a Low on BUSACK indicates that the CPU has relinquished the bus. If the buffered bus is shared, BUSACK must be used to control the latches and transceivers, as shown in Figures 4 through 6.
Address Latching (Demultiplexing the A/D lines)

The Z8002 uses a 16-bit time-shared Address/Data bus that must be demultiplexed, that is, latched for use with standard (not edge-activated) memories. AS is the obvious control signal for address latching and two LS373 Octal Transparent Latches are the best choice for this function (Figure 6). Note that addresses are not guaranteed valid when AS goes Low. It is therefore not possible to use the falling edge of AS to clock the addresses into edge-triggered registers. The rising edge of AS may be used as a clock, but this delays address availability by almost 100 ns. Transparent latches are the better choice.

Figure 6. Address Latches

ROM Addressing

Most microprocessors use nonvolatile memory for part of their program memory. Since the program status information for the Z8002 is read after Reset from locations 0002 and 0004, it is natural to use the lower half of the addressing space for ROM or EPROM.

This application uses 2716-type 2K x 8 EPROMs addressed by the latched addresses LA1–LA11. Pairs of 2716s store the low and high byte of each word. A0 is ignored since the Z8002 always reads a full word from memory. LA15 must be used as a Chip Select input to separate the ROM and RAM areas. When more than 2K words of ROM or EPROM are used, an LS138 one-of-eight decoder selects between the ROM and EPROM pairs.

When driven with a 4 MHz clock, the Z8002 requires a read access time (address valid from the CPU to data required into the CPU) of 400 ns. After subtracting a 27 ns propagation delay through the LS373 address latches and an 18 ns propagation delay through the LS243 transceivers, the ROM or EPROM must have an access time (address in to data out) of better than 355 ns. Some ROMs and EPROMs have a longer access time and therefore require an additional wait state that relaxes the access time requirement by an additional 250 ns. Figure 8 shows a 2-input NAND gate that generates a Wait signal whenever LA15 is Low and Q2 is High, thereby adding a wait state to every ROM/EPROM access.

RAM Address Multiplexing

Dynamic 16K x 1 RAMs such as the Z6116 provide read/write random-access storage. Sixteen of these devices populate the upper half of the addressable memory space (LA15 = High). Dynamic 16K RAMs use address multiplexing to reduce the package pin count, thus requiring only seven address inputs plus strobe inputs RAS and CAS.

Address Multiplexing. Two LS157 Quad Two-Input Multiplexers route the 14 address outputs LA1–LA14 into the seven RAM address inputs. MREQ synchronized with the rising clock edge is a convenient signal to control this multiplexer (Figure 7).
RAM Address Multiplexing (Continued)

**RAS and CAS Generation.** The address strobes RAS and CAS must be timed carefully with respect to the address information and the multiplexer control. Conceptually, MREQ might be used as RAS and DS as CAS. This would, however, require a memory read access time from the falling edge of CAS of approximately 120 ns (parameter 33 in the Z8001/Z8002 Product Specification Composite AC Timing Diagram, minus the 30 to 40 ns used by the CAS drivers and bus transceivers). Only the fastest 16K dynamic RAMs (the Zilog Z6116-2, for example) meet this requirement. Consequently, it is more practical to use a small amount of clocked TTL logic to generate earlier RAS and CAS signals and thus relieve the access time requirements so that even slow 16K RAMs (Z6116-3 and -4) can be used. Figure 8 shows the circuit that generates RAS and CAS.

RAS is a 2-clock period (500 ns at 4 MHz) wide active-Low signal starting on the falling clock edge when AS is Low. The address information is valid and stable during the specified hold time (<50 ns) immediately after the falling edge of RAS. RAS is generated by an LS109 edge-triggered dual JK flip-flop, clocked by CLOCK (that is, of a polarity opposite to the Z8002 clock). At the end of a machine cycle both Q1 and Q2 are High. The falling edge of CLOCK during AS clocks Q1 Low. The next falling clock edge leaves Q1 unaffected, but clocks Q2 Low. The next falling edge clocks Q1 High and leaves Q2 unaffected. The next falling clock edge clocks Q2 High and leaves Q1 High unless AS is Low, in which case the cycle is repeated. Q1 is Low from the center of the first to the center of the third T state. Q2 is Low from the center of the second to the center of the fourth T state.

The left half of the LS139 Dual One-of-Four Decoder generates CAS by ANDing three signals: LA15, MUX-S, and an auxiliary signal active during Read or DS.

During a read operation, CAS becomes active at the beginning of T2; that is, on the rising edge of CLOCK after MREQ has gone Low. During a write operation, CAS is delayed until the beginning of DS, when output data is guaranteed valid. The flip-flop stretches the width of DS, thus stretching CAS (during write operations) from 160 to 200 ns, as required by slower memories.

The right half of the LS139 decoder controls the routing of CAS to the two memory byte banks. The Z8002 addresses memory as bytes, but usually accesses words, ignoring A0. It uses A0 only when writing a byte, in which case it suppresses CAS to the byte bank that is not being written.

![Figure 8. RAS, CAS, and WAIT STATE Generators](image-url)
RAM Address Multiplexing
(Continued)

Figure 9. \( \overline{\text{RAS}} \) and \( \overline{\text{CAS}} \) Generation
Dynamic Memory Refresh. No external hardware is required for memory refresh. The Z8002 provides automatic memory refresh if properly initiated through a LDCTL instruction into the Refresh Control Register (Figure 10). Loading a 9E00 generates a refresh operation every 60 clock cycles (15 µs with a 4 MHz clock). This satisfies the worst-case refresh requirements of typical 16K dynamic RAMs.

Figure 11 shows the relationship between the upper byte of the refresh control register and the refresh period expressed in clock cycles. (Refer to the Z8000 PLZ/ASM Assembly Language Programing Manual, 03-3055-01 for further information.)

The Z8002 provides encoded status information on four outputs (ST0–ST3), which distinguish between three different interrupt acknowledge cycles; memory refresh; I/O reference; internal operation; data memory, stack or program memory access; and the first word of an instruction fetch. Two LS138 One-of-Eight Decoders can generate all the individual Status signals. For a simple system, only the first ten status codes have to be decoded. A single LS42 One-of-Ten Decoder is sufficient for this purpose (Figure 12).
Interfacing Peripheral Devices

Z-Bus compatible peripheral devices that require no external logic to interface with a Z8002 will become available in the near future. In the meantime, this application note describes the use of Z80A peripherals (PIO, CTC, SIO) with the Z8002. These peripherals require only a small number of additional TTL packages and a few lines of code to make the Z8002 emulate the typical Z80A control signals IORQ, M1, RD, and RETI.

Four different operations are performed between the CPU and its peripherals:
- CPU writing into the peripheral device.
- CPU reading from the peripheral device.
- Peripheral interrupting the CPU, which responds with an Interrupt Acknowledge.
- CPU issuing a Return from Interrupt (RETI) signal.

The first two operations—writing to or reading from the peripheral—are fairly straightforward. An LS138 One-of-Eight Decoder, enabled by the decoded Status signal IORQ, decodes the latched I/O address and generates CE signals to the individual peripheral devices (Figure 13). The Z8002 can use the full 16-bit address space for I/O, but this application uses only LA3-LA10. When necessary, the higher-order address bits can also be decoded and fed into one of the Enable inputs. Notice that all ports require an odd address to interface to the Z8000 data bus (lower byte).

A write operation into the enabled peripheral is performed when IORQ is Low while RD is High. Similarly, a read operation from the enabled peripheral is performed when IORQ is Low while RD is Low.

![Figure 13. Z8002 to Z80A Control Translation](image-url)
Interfacing Peripheral Devices (Continued)

The first four I/O devices addressed when LA0 is Low are four Z80A peripheral components. The fifth peripheral is a set of eight switches that can be read by the CPU, which addresses them as a peripheral device. The user can thus specify any one of 256 different conditions (for example, choosing between 16 different baud rates for each of the two serial I/O channels). The sixth CE output addresses a phantom peripheral called RET!, which is activated at the end of an interrupt service operation.

The interrupt operation requires some extra logic and software to make the Z80A peripherals compatible with the Z8002. Z80A peripherals request a vectored interrupt by pulling the VI input of the CPU Low. The CPU (Z80A or Z8002) samples this input at a specified time prior to the end of any instruction execution. The Z8002 then acknowledges the interrupt with a specific Status code (VIACK). The Z80A, which has no dedicated Interrupt Acknowledge output, acknowledges interrupts by issuing a unique combination of control signals: IORQ active during an MI cycle (MI normally indicates the opcode fetch cycle of an instruction execution). Z80A peripherals resolve potential conflicts between overlapping interrupt requests from different interrupting devices by means of a daisy-chain arrangement between the IEO outputs and the IEI inputs of the peripheral components. The highest-order peripheral has its IEI permanently tied High. For any peripheral that has no interrupt pending or under service, IEO = IEI. Any peripheral that has an interrupt pending or under service forces its IEO Low.

To insure stable conditions in the daisy chain, all interrupt status signals are prevented from changing while MI is Low. When IORQ is Low, the highest priority interrupt requestor (the one with IEI High) places its interrupt vector on the data bus and sets its internal interrupt-under-service latch. Figure 14 shows the Interrupt Acknowledge timing.

The circuit shown in Figure 13 generates the MI, IORQ, and RD signals required by the Z80A peripherals during an interrupt acknowledge cycle.

**Return From Interrupt.** At the end of an interrupt service routine the interrupt-under-service latch in the Z80A peripheral that has been serviced must be reset. The Z80A CPU accomplishes this by executing a special 2-byte instruction with the opcode sequence ED-4D (RETI) appearing on the data bus. All peripherals monitor this sequence and manipulate the daisy chain to reset the appropriate internal interrupt-under-service latch. The normal daisy-chain operation can be used to detect a pending interrupt; however, it cannot distinguish between an interrupt under service and a pending unacknowledged interrupt of a higher priority. Whenever "ED" is decoded, the daisy chain is modified by forcing High the IEO of any interrupt that has not yet been acknowledged. Thus the daisy chain identifies the device presently under service as the only one with an IEI High and an IEO Low. If the new opcode byte is "4D," the interrupt-under-service latch is reset (Figure 15).

The Z8002 does not have the equivalent RETI instruction and must therefore simulate it with a combination of hardware and software. A software sequence at the end of every interrupt service routine writes two consecutive bytes.
Interfacing Peripheral Devices (Continued)

(ED followed by 4D) into the phantom peripheral called RETI. The recommended software sequence is as follows:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI</td>
<td>Disable Interrupts</td>
</tr>
<tr>
<td>LDB</td>
<td>Load First Byte</td>
</tr>
<tr>
<td>OUTB</td>
<td>Output First Byte</td>
</tr>
<tr>
<td>LDB</td>
<td>Load Second Byte</td>
</tr>
<tr>
<td>OUTB</td>
<td>Output Second Byte</td>
</tr>
<tr>
<td>EI</td>
<td>Enable Interrupts</td>
</tr>
<tr>
<td>RET</td>
<td>Return From Interrupt</td>
</tr>
</tbody>
</table>

To prevent the two byte simulated RETI instruction from being interrupted, interrupts must be disabled. If NMI's can occur at any time, then interrupts must remain disabled throughout the NMI service routine. This allows the Z80A peripheral devices to decode correctly a RETI instruction. During the two OUTB operations, each four clock cycles long, RETI is Low, VIACK is High and the Z80A control signals MI and RD are Low.

Driving Z80A Peripherals. The Z80A PIO, CTC and SIO are directly connected to the appropriate lines, as follows. The bidirectional AD0-AD7 buffers are connected to the D0-D7 data inputs/outputs on the peripherals.

- The address bits LA1 and LA2 are used as Port Select (A/B) and Control Data select (C/D) on the PIO and SIO, and as Channel Select (CS0, CS1) on the CTC.
- The Interrupt outputs of all peripherals are interconnected (pulled up with a 4.7kΩ resistor to VCC and connected to the VI input of the Z8002). The IEI-IEO interrupt daisy chain of the Z80A peripheral devices must be connected appropriately to establish the desired hierarchy of interrupt priorities.
- The Z80A PIO requires a M1 to enable the peripheral circuit's internal interrupts. This can easily be accomplished by writing a dummy byte (00H) to the RETI port after PIO interrupts have been enabled.

Reset

The Z8000 Reset input requires a minimum High level of 2.4 V. While TTL High levels are guaranteed to be at least 2.4 V, this does not leave margin for noise immunity. If an open collector buffer (such as a 7407) is available, an output pullup resistor to +5 V will provide more than adequate margin for noise immunity. If an open collector gate is not readily available, a standard TTL gate may be used with an output pullup resistor. In this case, the value of the pullup resistor should not be less than 300 Ω.

Conclusion

This Application Note demonstrates that a small, but powerful computer can be built around the Z8002 16-bit microprocessor using very few standard TTL support packages. It also shows how the readily-available Z80A peripheral circuits interface easily to the Z8002, taking advantage of the similarity in the Z80A and Z8000 interrupt structures.
**An Introduction to the Z8010 MMU Memory Management Unit**

**Tutorial Information**

March 1981

**Introduction**

The declining cost of memory, coupled with the increasing power of microprocessors, has accelerated the trend in microcomputer systems to the use of high-level languages, sophisticated operating systems, complex programs and large data bases. The Z8001 microprocessor supports these advances by offering multiple 8M byte address spaces as well as a rich and powerful instruction set. The Z8010 Memory Management Unit (MMU) supports the Z8001 processor in the efficient and flexible use of its large address space.

Support for managing a large memory can take many forms:

- Providing a logical structure to the memory space that is largely independent of the actual physical location of the data
- Protecting the user from inadvertent mistakes such as attempting to execute data
- Preventing one user from unauthorized access to memory resources or data
- Protecting the operating system from unexpected access by the users.

The Z8010 provides all these features plus additional features that permit a variety of system hardware configurations and system designs.

This paper examines the various uses of memory management in computer systems and how memory management techniques generally meet these requirements. The major features of the Z8010 MMU illustrate how memory management functions can be supported by hardware. A few examples demonstrate how this LSI circuit can be used to configure several different memory management systems.

**Motivations for Memory Management**

The primary memory of a computer is one of its major resources. As such, the management of this resource becomes a major concern as demands on it increase. These demands can arise from different sources, three of which are of interest in the present context. The first stems from multiple users (or multiple tasks within a dedicated application) contending for a limited amount of physical memory. The second comes from the desire to increase the integrity of the system by limiting access to various portions of the memory. The final source arises from issues surrounding the development of large, complex programs or systems. Each of these three sources involves a multifaceted group of related issues.

When multiple tasks constitute a given system (for example, multiple users of a system or multiple sub-tasks of a dedicated application), the possibility exists that not all tasks may be in primary memory at the same time. (A task is the action of executing a program on its data; a task may be as simple as a single procedure or as complex as a set of related routines.) If the population of memory-resident tasks can vary over time, a useful feature of a system would be the ability for a task to reside anywhere in memory, and perhaps in several different locations during its lifetime. Such tasks are called relocatable, and a system in which all tasks are relocatable generally offers greater flexibility in responding to changing system environments than a system in which each task must reside in a fixed location.

A second issue that arises in multi-task environments is that of sharing. Separate tasks may execute the same program on different data, and may therefore share common code. For example, several users compiling FORTRAN programs may wish to share the compiler rather than each user having a separate copy in memory. Alternatively, several tasks may wish to execute different programs using the same data as input, and it may be possible for these tasks to access the same copy of the
Memory Management (Continued)

input. For example, a user may wish to print a PASCAL program while it is being compiled; the print process and the compiler process could access the same copy of the text file.

A third issue in multi-task systems is protecting one task from unwanted interactions with another. The classic example of unwanted interaction is one user's unauthorized reading of another user's data. Prohibiting all such interactions conflicts with the goal of sharing and so this issue is usually one of selectively prohibiting certain types of interactions. The issue of protecting memory resources from unauthorized access is usually included in the larger set of issues relating to system integrity.

System integrity takes many forms in addition to protecting a task's data from unwanted access. Another aspect is preventing user tasks from performing operating system functions and thereby interrupting the orderly dispatch of these tasks. For example, most large systems prevent a user task from directly initiating I/O operations because this can disrupt the correct functioning of the system.

Another aspect of separating users from system functions relates to separating system I/O transfers from user tasks, especially with respect to error conditions. For example, an error during a direct memory access, say to a nonexistent memory location, should not cause an error in the program that is currently executing.

A final example of increasing the system integrity is protecting a user task from itself. Obvious errors, such as trying to execute data or overflowing an area set aside for a stack, can be detected while a program is executing and handled appropriately, provided the system is given sufficient information.

The notion of protecting an executing task from performing certain types of actions known to be erroneous introduces a third general motivation for memory management, namely support for the design and correct implementation of large, complex programs and systems.

Protecting a task from itself obviously helps in debugging a large program, but there are other system features that can aid in developing complex systems. Modern methodology for developing large systems dictates partitioning a task into a number of small, simple, self-contained sub-tasks with well defined interfaces. Each sub-task generally interacts with only a few other sub-tasks and this communication is carefully controlled. This methodology promotes a systems design that can be readily modified, but it also tends to promote the creation of a large number of nearly independent sub-tasks and many data structures accessible to only one or a few of these sub-tasks.

Because modern systems are increasingly driven to support many interacting tasks, possibly written and compiled separately, they must also enforce some communication protocol without sacrificing efficient operation. Modern memory management systems can offer effective tools for implementing large systems designed using this methodology.

In summary, the major goals of memory management systems are to:

- Provide flexible and efficient allocation of memory resources during the execution of tasks
- Support multiple, independent tasks that can share access to common resources
- Provide protection from unauthorized or unintentional access to data or other memory resources
- Detect obviously incorrect use of memory by an executing task
- Separate users from system functions

Most of today's memory management systems support these functions to some degree. The extent of this support is largely a question of resources to be devoted to these functions and the understood demands of the intended applications for these systems.

The Fundamentals of Memory Management

Memory management has two functions: the allocation and the protection of memory. Dynamic relocation of tasks during their execution is accomplished by an address translation mechanism. The restriction of memory access is accomplished by memory attribute checking. Both operations occur with each memory request during the execution of a program and both are transparent to the user.

Address translation simply means treating the memory addresses generated by the program as logical addresses to be interpreted or translated into actual physical memory locations before dispatching the memory access requests to the memory unit. Memory attribute checking means that each area of memory has associated with it information as to who can access it and what types of access can be made by each task. Each memory reference is checked to insure that the task has the right to access that location in the given fashion (for example, to read the contents of the location or to write data to that location).

Instead of a linear address space, more elaborate memory management systems have a hierarchical structure in which the memory consists of a collection of memory areas, called segments. Access to this structured memory requires the specification of a segment and an offset within that segment. Thus, instead of specifying memory location 1050 in a linear address space, a task specifies memory location 5 in segment number 23, for example.
Generally, segments can be of variable size, within limits, and a user can specify the size of each segment to be used. Thus one user may have two segments of two thousand and ten thousand words for his FORTRAN program and data, respectively, while another user might have three segments of three thousand, six thousand and two thousand words for her PASCAL program, data, and run-time stack. If the first user called his data segment number 5, then the first word in his data set would be accessed by the logical address (5,0) indicating segment 5, offset 0. The memory management system translates this symbolic name into the correct physical memory address.

Figure 1 gives a conceptual realization of these two users' logical program spaces. The first user, User A, has his program segment called "Segment 6" and his data segment called "Segment 5." The second user, User B, has her program segment called "Segment 5," her data segment called "Segment 12" and her stack segment called "Segment 2." Notice that both users have named one of their segments "Segment 5," but they refer to different entities. This causes no problem since the system keeps the two memory areas separate. The situation is analogous to both users having an integer variable called "I" in their programs: The system realizes that these are two separate variables stored in different memory locations.

User A's data segment, "Segment 5," is ten thousand words. If he references word 10,050 of Segment 5 he gets an error message from the system indicating that he has exceeded the allocation limit for Segment 5. Note that he does not access word 50 of Segment 6. That is, segments are logically distinct and unordered. A reference to one segment cannot inadvertently result in access to another segment. Thus, in this example, User A is prevented from accidentally (or deliberately) accessing his program as though it were part of his data segment.

Figure 2 illustrates one way that these segments could be arranged in the physical memory. The dotted lines indicate the memory-mapping function from the logical address space of the user to the physical memory locations allocated to him. The figure also indicates the access attributes associated with each user's segments. For example, program segments are "execute only" and data segments are "read/write." Thus a user is prevented from executing a data segment or writing into a code segment.

![Figure 1. Two User's Logical Address Space](image1.png)

![Figure 2. Mapping Logical Segments to Physical Memory](image2.png)
The Fundamentals of Memory Management (Continued)

Figure 3 illustrates what happens when both users have access to the same data set in primary memory, say the results of a questionnaire that both intend to analyze. Each user has a logical name associated with that data set to specify the segment in which the data set is to reside. Note that the two users have chosen to put the data set in different segments of their personal address spaces. The system-mapping function translates these different segment names to the same physical memory locations. Thus User A’s access to address (2, 17) references the same physical memory location as User B’s access to address (7, 17). In the figure, note that two of B’s segments have been moved in physical memory to create a space large enough to hold the questionnaire data.

Another topic in memory management that is supported by Z8001-Z8010 architecture but requires additional support hardware is demand swapping, or segmented virtual memory, which means that the logical memory area may not actually reside in physical memory until a task actually tries to access it. At the time an access is made to a segment missing from physical memory, the instruction execution is held in abeyance until the logical memory can be brought into the physical memory and then the instruction is allowed to proceed with the memory access. The address translation is performed, access protection is checked and the instruction proceeds as if the logical memory area had been in the physical memory at the beginning of the instruction. The instructions in the Z8001 must run to completion before the CPU can perform any action, such as responding to a missing segment trap. But with the conjunction of hardware and software to simulate the above functions, a segmented virtual memory scheme can be implemented.

A final topic in memory management is paging, which is another method for partitioning a user address space and mapping it onto the physical memory. Paging is most effective when demand swapping can be supported. Essentially, paging divides the logical memory into fixed-size blocks, called pages. Like segments, the individual pages can be located anywhere in the physical memory and a translation mechanism maps logical addresses to physical memory locations. There are two differences between paging and segmenting a logical memory. First, pages are of fixed size whereas segments are of various sizes. Second, under paging, the logical memory is still linear, that is, a task accesses memory using a single number, rather than a pair as in segmentation. The major advantage of paging is in treating memory as blocks of fixed sizes, which simplifies allocating memory to users and deciding where to place the logical pages in physical memory. The major disadvantage of paging is in assigning different protection attributes to different areas in a user address space because a paged memory appears homogeneous to the user and the operating system. Paging can be combined with segmentation to produce a memory management system with the advantages of both paging and segmentation. The implementation of paging for the Z8001 requires additional support hardware and may be implemented independent of the Z8010.

Before proceeding to the mechanism of memory management, it is instructive to review how a segmented address translation mechanism with protection attributes achieves the five major goals of memory management outlined in the previous section. The first goal permits dynamic allocation of memory during the execution of tasks; that is, a task could be located anywhere in memory and even moved about when its execution is suspended. The address translation mechanism provides this flexibility because the task deals exclusively...
with logical addresses and hence is independent of the addresses of the physical memory locations it accesses. Moving the task to different physical memory locations requires that the address mapping function be changed to reflect the change in memory location, but the task's code need not be modified. Of course, this flexibility does incur the price of managing the various system tables required to implement memory management.

The second goal supports sharing of common memory areas by different tasks. This is accomplished by mapping different logical areas in different tasks to the same physical memory locations.

The third provides protection against certain types of memory accesses. This is accomplished by associating accessing attributes with each logical segment and checking the type of access to see if each access is permitted.

The fourth goal detects obvious execution errors related to memory accessing. This can be accomplished by checking each access to a segment to see whether the address falls within the allocated physical memory for that segment. It could also include affixing a read/write attribute to data to prevent a task from trying to execute a data segment, and affixing an execute-only attribute to code segments to prevent a task from trying to read or write data to this segment. Additionally, if a segment is used for a stack, the system could issue a warning to a task when the stack approaches the allocated limit of the segment. The task could then request more memory for the stack before the stack overflows and creates a fatal error.

The final goal listed for memory management systems separates user functions from system functions. For processors that distinguish between System mode and User mode of operation, this goal can be accomplished by associating a system-only attribute with system segments so users cannot directly access system tables and tasks.

As a final point, it should be noted how segmentation can be used to support the development and execution of large, complex programs and systems. The concept of segmentation corresponds to the concept of partitioning a large system into procedures and data structures where each procedure and data structure can be associated with a separate segment. A task can then invoke a procedure or sub-task or access a data structure by referring to its logical segment name. Access to these objects can be individually restricted by using the protection-checking mechanism of the memory management system.

As a specific example of how segmentation could be used in the design of a large system, consider a multi-user interactive BASIC system with a large data base shared by all users. Such a system could be designed with segments 0 through 15 reserved for system use, segments 16 through 31 reserved for the BASIC interpreter and its internal tables, segments 32 through 63 allocated to user tasks and segments 64 through 127 reserved for portions of the data base when they are in primary memory being accessed by users. For this system, segments 0 through 31 would probably always be in memory; the other segments would be assigned as needed and the memory they require allocated dynamically.

Essentially there are four issues in implementing a memory management system: how addresses are specified, how these addresses are translated, what attributes are checked for each access, and how the protection mechanism is implemented. Some of the major alternatives in each of these issues are briefly discussed here, primarily from the point of view of a segmented memory.

Two approaches have traditionally been taken for specifying addresses in a segmented memory. For simplicity, only addresses in instructions are discussed. The first way puts all the addressing information in the instruction itself. That is, each memory address in an instruction contains both the segment name and the offset within the segment. The alternative sets aside special registers that contain some of this information, for example the segment name or the address in physical memory where the segment resides.

The advantage of the latter approach lies in the fact that fewer bits are needed in an instruction to specify addresses. Thus programs may be shorter. Also, because there is reduced traffic between the memory and the processor for fetching shorter instructions, a program may execute faster.

On the other hand, these special registers must be manipulated to access more segments than there are registers, and this manipulation adds to the number of instructions, the program size and the execution time. In practice, these can destroy the advantages described above. If the special registers contain physical memory locations, then these must be protected from user access to maintain the integrity of the system, and changing segments requires system calls which can be time consuming if too few registers are supplied. The Z8001 architecture specifies the complete logical address in the instruction.

Address translation is performed by adding the logical segment offset to the memory location where the segment begins. Thus, when an address of the form (a, b) is presented to the translation mechanism, the segment name "a" is used to determine where segment "a" resides in memory. Assume that it resides in locations 10000 to 25000. Then the actual
The Mechanics of Memory Management
(Continued)

The memory location of \((a, b)\) is memory location \(10000 + b\). The major option in implementing this type of address translation is in determining the segment location in physical memory. When special registers have been set aside to contain the starting location of the segment instead of putting all address information in the instruction, the addressing mechanism is similar to using the segment register as an index register or a base register.

When logical addresses are either completely specified in the instruction or when the special register contains the symbolic segment name, a table must be used to translate the logical segment name into a physical memory location. The table may have an associative capability, that is, the segment name is presented to the table and the device returns the physical memory location where the segment begins. Alternatively, the table could have one entry for every possible segment name. The Z8010 implementation of the address translation table sets aside a specific table entry for each logical segment name.

A number of attributes can be associated with a segment and checked during each access. One of these is the allocated length of the segment, and each access is checked to see if it falls within the bounds of the segment. The Z8010 provides limit checking.

Another type of attribute deals with ownership or class of ownership: tasks are grouped into classes and only those in certain classes are permitted access. The simplest example is the system versus user classification, where tasks are either one or the other and this determines whether or not any type of access can be made to the segment. The Z8010 has this feature—users are prevented from accessing system segments.

Other types of attributes that can be associated with a segment involve modes of accessing, for example read only, read/write or execute only. For these attributes, the processor must indicate the type of access to be made, be it code fetch, read from memory, write to memory, etc. The Z8001 indicates when it is fetching code, reading or writing data, or performing stack operations, and thus the Z8010 can offer protection for these operations. The other issue with respect to attributes is whether they are permissive or prohibitive. That is, whether the attribute is in the form of "write to this segment is permitted" or of the form "write to this segment is prohibited." The Z8010 adopts the approach of specifying attributes that prohibit certain types of accessing.

The final issue in the mechanics of memory management systems is the implementation of the protection attributes. These may be associated either with the logical address space or with the physical memory itself. The IBM 360 series, for example, places the memory protection information with the physical memory itself. Thus the processor generates a memory address and the memory module checks to see if the access is permitted. The main difficulty with this approach is in the lack of flexibility, because protection is associated with fixed memory partitions. Also, sharing memory is cumbersome because each user is given a protection key to match the memory key; thus both users must have the same access key or a universal access key.

Associating access attributes with the logical segment permits a versatile memory management scheme because different users can access the same segment and have different access attributes associated with their accessing. The Z8010 implements access attributes using the segment mapping information.

Other information associated with each segment does not pertain to the protection mechanism but can be of use to the memory management system. This information generally relates to the history of the segment; for example, whether a segment has been modified while resident in primary memory. If it has not been modified and the system requires the memory for another segment, the memory can be freed immediately; otherwise, the updated version of the segment must be stored in secondary memory and the primary memory is not available until the segment has been saved. Although not strictly necessary, such information can improve the performance of the memory management system. The Z8010 collects information on segment usage, and this information can be used to enhance performance of systems that use this device.

<table>
<thead>
<tr>
<th>The Z8010 Memory Management Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Z8001 CPU generates segmented addresses consisting of a 7-bit segment number and a 16-bit segment offset address. In addition, the CPU generates status signals indicating its current mode of operation (such as Instruction Fetch, Data Memory Reference, Stack Memory Reference, and Internal Operation), whether it is performing a Read or a Write Memory Reference and whether it is in Normal (User) or System Mode. The Z8010 Memory Management Unit uses this information to perform its memory management functions. This section describes the Z8010 MMU in some detail, beginning with the translation procedure and continuing with a description of the internal registers of the chip. The section concludes with a description of the system commands that alter the contents of these registers.</td>
</tr>
</tbody>
</table>

The Z8010 MMU has three functional states. The first is the memory management state: when a logical address is presented to the unit, the MMU checks the access to insure its validity and translates the logical address to a physical memory location. The second state is a command state: when a special I/O instruc-
The Z8010
Memory
Management
Unit
(Continued)

The Z8010
Memory
Management
Unit
(Continued)

The inputs to the MMU are the Address/Data lines (A/D lines), Segment Number lines, Bus Status and Timing Lines, and special control lines for chip selection and DMA. The outputs from the MMU are Address lines, a Segment Trap line and a Suppress line (Figure 4). During address translation and access protection, logical addresses are presented to the MMU on the Segment Number and Address/Data lines; the MMU puts the translated physical memory location on its Address lines and, if appropriate, activates the Segment Trap and/or Suppress lines.

Segment Trap is a special type of synchronous interrupt for the Z8001 CPU; Suppress aborts the memory access. In the command state, the MMU receives commands on the A/D lines; data to be read from or written into the MMU is also placed on the A/D lines.

The MMU selects which of the three states it will be in according to the status information on the Bus Status lines during the initial clock cycle of an instruction or DMA cycle. The MMU performs address translation during a memory reference for either a regular instruction or a DMA request. Only I/O instructions (either regular or special), memory refresh and reserved bus status states cause the MMU to cease performing memory address translations and enter another state.

The MMU uses the segment number to access an internal table of segment descriptor registers, each register containing the starting memory location of the segment (called the base address), the segment’s limit (used to determine the range of legal address offsets) and the types of accesses permitted to that segment.

Physical memory for segments is allocated in blocks of 256 bytes. The eight least significant bits of the base address are all zero and are not stored in the Segment Descriptor Register. Also, since the eight low-order bits of the segment base are always zero, the eight low-order bits of the segment offset need not participate in the addition of the base address to the offset. Rather, they can be juxtaposed to the result of adding the high-order byte of the offset to the most significant 16 bits of the base address.

This process is illustrated in Figure 5. Note that the low-order eight bits of the offset are not used by the MMU. Figure 6 goes through an example of mapping the logical address (5, 1528) to a physical memory location when segment 5 begins at location 231100.

Figure 6a illustrates the full addition to be performed during address translation. The segment number 5 selects Segment Descriptor Register 5 in the MMU. The base address field in this register contains 2311 which corresponds to a base address of 231100. The offset, 1528, is then added to 231100 to produce the physical memory location 232628. Figure 6b represents the same logical procedure, but illustrates the actual operation of the MMU. Again segment number 5 is used to select the base address. However, only the high-order byte of the offset is added to the contents of the base address.
The Z8010 Memory Management Unit (Continued)

The Z8010 Memory Management Unit (MMU) is used to create a physically addressable virtual memory space for programs and data. It works in conjunction with the CPU to translate physical addresses to virtual addresses and vice versa. The MMU contains several registers, including the Segment Descriptor Registers, which are used to define segments of memory.

MMU Registers

- **Segment Limit Field**: Specifies the maximum physical address for a segment.
- **Segment Base Field**: Specifies the base address of the segment.
- **Segment Type Field**: Specifies the type of segment, such as code, data, or I/O.
- **Permissions Field**: Specifies read, write, execute, and other permissions for the segment.
- **Base-Address Field**: Used to store the base address of a pointer or stack segment.
- **Index Field**: Used to store an index for a pointer or stack segment.
- **Page Mode**: Specifies if the segment is in page mode or linear mode.
- **Page Offset**: Specifies the offset within a page.
- **Segment Offset**: Specifies the offset within a segment.

Control Flags

- **Abort Flag**: Indicates that the current instruction has been aborted due to an error.
- **VIOLATION Flag**: Indicates that a violation has occurred.
- **Suppression Flag**: Indicates that the current instruction is being suppressed due to an error.
- **Warning Flag**: Indicates that a warning has occurred.
- **Fatal Flag**: Indicates that a fatal error has occurred.

The MMU checks memory references for two types of trap conditions. The first type is an access violation. This occurs when a memory reference is performed in a mode that is not allowed by the read-only, execute-only, CPU-inhibit or system-only attribute of a segment. A memory reference outside the allocated memory for the segment also constitutes an access violation.

The second type is a write warning. This occurs when a write is made to the last 256 bytes of a special type of segment (indicated by a special attribute flag called the Direction And Warning Flag). These segments are typically used for stacks and are therefore logically organized so that successive writes (or stack pushes) access lower-numbered memory locations. By generating a segment trap request when a write is performed into the lowest-numbered 256 bytes of the memory allocated for these segments, the MMU is signaling that a stack is in danger of overflowing. The operating system in servicing this trap can increase the memory allocated for the segment and avoid a fatal stack overflow condition.

The MMU generates two control signals that can be used by the system to perform memory management functions. Segment Trap Request is generated upon the first detected occurrence of a violation or write warning. Once asserted, this signal remains set until a trap acknowledge signal is received. Only when the Fatal Flag, a special MMU control flag, is set will a detected violation not cause a segment trap request. This flag is set only when a second violation is detected while a previous trap is being processed and thus indicates that the system software is in error.

The other control signal generated by the MMU is Suppress. Once a violation has been detected, this signal is asserted on that and every succeeding memory reference for the remainder of the instruction. In particular, I/O and Special I/O instructions are checked for memory access violations, and once a memory access violation is detected, subsequent memory accesses cause Suppress signals to be generated. I/O addresses, of course, bypass the MMU and are neither translated nor checked. Intervening DMA cycles and memory refresh cycles are exceptions to this rule. During such cycles Suppress is not asserted unless a violation is detected during that cycle. Only DMA can generate a violation; refresh can never cause a violation. Suppress can be used by the memory system to inhibit writes, thus protecting the memory from illegal alterations.

Because there are 64 Segment Descriptor Registers in the MMU, two MMUs are required to handle all 128 segments that the Z8001 can manipulate directly. An MMU is programmed to handle either segments 0 through 63 or segments 64 through 127; the particular set of 64 segments in an MMU can be changed using special operating system commands. Each Segment Descriptor contains three fields, a 16-bit Base Field, an 8-bit Limit Field and an 8-bit Attribute Field (Figure 7). The segment number of a logical address determines which segment descriptors are used in address translation.

The **Base Field** specifies the starting location in memory of the segment.

The **Limit Field** specifies the segment size in blocks of 256 bytes. The address offset is compared against the segment limit and a size violation occurs if the offset falls outside the segment boundaries. A write warning occurs if the destination is in the last block of a segment being used as a stack.

<table>
<thead>
<tr>
<th>Figure 6. Two Methods of Address Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Address Translation Diagram]</td>
</tr>
<tr>
<td>a) FULL ADDITION</td>
</tr>
<tr>
<td>b) ADDITION OF HIGH ORDER BYTES ONLY</td>
</tr>
</tbody>
</table>

---

3-36
The Attribute Field contains eight flags. Five flags protect the segment against certain types of access, one indicates a special orientation of the segment, and two indicate the types of accesses that have been made to the segment. The following brief description explains how these flags are used.

The Read-Only Flag (RD) indicates that the only accesses to this segment are reads. Writes are prohibited when this flag is set. Thus this flag is a write-inhibit flag; in particular, code can be executed from a read-only segment. This flag is useful in protecting data from being written by unauthorized users. For example, if one user wants to give another access to a document that he has created, but does not want this user to be able to modify it, the system can set the Read-Only Flag when it copies the file into the user’s address space. If the data is already in memory (in a read-only mode), then this same memory area can be made accessible to that user without another copy of the document being required.

The System-Only Flag (SYS) indicates that only accesses made in System Mode are to be permitted. When this flag is set, accesses in the Normal Mode are prohibited. This attribute is useful in protecting system tables and tasks from being accessed by users. For example, system I/O routines can be left in the memory with this flag set and a user is unable to call them directly. This feature is useful if a system is designed so that users are given certain segment names and other segment names are reserved for system use. This flag prevents users from accessing system segments, even though they can generate the logical addresses.

The CPU-Inhibit Flag (CPUI) indicates that the segment is not to be referenced by the CPU. When this flag is set, CPU access to this segment is prohibited, but DMA channels can access the segment. This flag is useful in preventing a program from accessing a segment whose data resides on secondary storage and has not been brought into primary memory. For example, a user may request the operating system to read a file from disk into segment number 19; if the operating system returns control to the user before the file has been read, this flag should be set in Segment Descriptor Register 19.

The Execute-Only Flag (EXC) indicates that the segment is to be referenced only during the instruction fetch cycle of the processor. When this flag is set, access to the segment during any other cycle of an instruction, for example during the memory request cycle, is prohibited. This flag is useful in preventing a program from making a copy of a proprietary program. For example, if this flag is set for a segment containing code that a user can access, that code is protected from being read and hence from being copied.

The DMA-Inhibit Flag (DMAI) indicates that the segment is not to be referenced by a DMA Channel. When this flag is set, only the CPU has access to the segment. This flag is useful in preventing a DMA device from modifying a segment being used by an executing task. For example, segments with valid data should have this flag set to protect them from modification by a DMA device.

The Direction And Warning Flag (DIRW) indicates that memory accesses are to be monitored and certain accesses are to be signaled, although allowed to proceed. When this flag is set, any write to the lowest 256 bytes of the segment generates a write warning. This flag is useful for segments that are used as stacks since the Z8001 has special stack instructions to manipulate stacks that grow toward lower memory locations. Thus a write warning for a stack indicates that the stack may soon overflow its allotted memory space and that more physical memory should be obtained. For example, if a segment serves as a run-time stack for a block-structured programming language such as PASCAL, memory can be allocated to this segment only as a program requires during its execution. The alternative in a fixed allocation environment is to allocate as much memory for the stack as the system expects the program to need, whether or not it is actually used by the program.

The Changed Flag (CHG) indicates that a write has occurred to this segment. This flag is set automatically whenever a program or DMA device writes into the segment. This flag is useful in indicating which segments have been modified in the case where the segment must be written to a secondary storage device. Segments that have not been updated need not be copied back to disk if a copy already exists. For example, when a user task is suspended in a multiple-user environment and his task is to be swapped out of memory temporarily to make room for another task, only those segments that have been changed need to be updated on the disk.

The Referenced Flag (REF) indicates that a memory access has been made to a segment. This flag is set automatically whenever a program or DMA device accesses the segment. This flag is useful in indicating which segments are active in the case that a segment must be
selected to be swapped out of primary memory to make room for another task. For example, seldom-used operating-system tasks that usually reside in primary memory may be swapped out to make room for users with large memory requirements. This flag is a way of ascertaining which segments contain seldom used tasks.

Three user-accessible 8-bit registers in the MMU control the functioning of the MMU (Figure 8). The Mode Register provides a sophisticated method for selectively enabling MMUs in a multiple-MMU configuration. The Segment Address Register (SAR) selects a particular segment descriptor to be accessed by a system routine when it is changing the organization of primary memory. The Descriptor Selection Counter Register selects the particular byte in the Segment Descriptor Register that is accessed.

Two flags in the Mode Register govern the functioning of the MMU. The Master Enable Flag (MSEN) indicates whether the device will perform address translation. When this flag is set, addresses translated by the MMU are placed on its Address lines; when this flag is clear, the Address lines are 3-stated. Thus, once this flag is reset, no memory request can pass through the MMU. In a single-MMU configuration, MSEN set to zero requires that the CPU must have access to a special memory, since it will not be able to fetch an instruction from the primary memory. This flag can be set during hardware reset (this is discussed later).

The second flag in the mode register that governs the functioning of the MMU is the Translate Flag (TRNS). This flag indicates whether the MMU is to translate the addresses presented to it. When the flag is set, the MMU translates logical addresses to physical memory locations and checks to see if a violation will occur on that access. When the flag is clear, addresses presented to the MMU are passed to the output Address lines without change, and no protection checking is done.

When multiple-MMUs are used in a memory-management system, some mechanism must be present to select those devices that are to be active during the memory translation process. More specifically, if two MMUs are employed so that all 128 segments can be used at random by an executing process, then some way must exist for each of the MMUs to know which 64 Segment Descriptors are located in its Segment Descriptor Registers. The Upper Range Select Flag (URS) indicates which set of 64 descriptors is stored in the MMU. When the flag is set, the MMU contains descriptors 64 through 127; when the flag is reset, the MMU contains descriptors 0 through 63.

When multiple-MMU devices keep separate tables for system descriptors and user descriptors, the Multiple Segment Table Flag (MST) and the Normal Mode Select Flag (NMS) in the Mode Register distinguish which MMUs contain system descriptors and which contain user descriptors. When the MST flag is set, multiple tables are present in the configuration, and each MMU is dedicated to one of the tables. In this case the MMU translates addresses only when the N/S signal matches the NMS flag. Thus, if there are two tables in the memory management system (one for the system and one for users), the NMS flag is set in those MMUs containing the users' segment descriptors, and is not set in the remaining MMUs. All MMUs in the system have the MST flag set to indicate more than one table in the system.

The final piece of control information in the Mode Register is a 3-bit Identification Field (ID) that indicates a logical name for the MMU. When a segment trap is acknowledged by the CPU, the MMU uses this field to select one of the A/D lines; each enabled MMU should select a different line. If an MMU requested a segment trap, it outputs a 1 on its assigned A/D line; otherwise it outputs a 0. Since the ID field is three bits, up to eight MMUs can be uniquely identified. One instruction might result in multiple violations in different MMUs, so that the segment trap software might have to deal with several MMUs to process the trap.

The other two control registers in the MMU are the Segment Address Register (SAR), which points to one of the 64 segment descriptors, and the Descriptor Selection Counter Register. Commands to read or write a segment descriptor use the SAR pointer to select which descriptor is to be accessed. This register has an auto-incrementing capability for accessing consecutive descriptors in succession without having to reload the SAR. Thus if descriptors 0 through 4 are to be modified, the SAR is initialized to 0 and then auto-incremented to point to descriptors, 1, 2, 3 and 4 in succession.

The Segment Descriptor Number is a 6-bit field that contains the address of the descriptor within the MMU. If the MMU holds segments 64 through 127 (that is, if the URS flag is set), the segment named 64 is accessed when the SAR number field is 0. This is a result of the 6-bit limit of the descriptor number field. The field indicates the 6 least-significant bits of the logical segment descriptor number.
Segment Descriptors consist of four bytes; the Descriptor Selection Counter indicates which byte is being accessed during a command (commands to the MMU can read or write only one byte at a time). A counter value of 0 indicates the high-order byte of the base address is being accessed, 1 indicates the low-order byte of the base address, 2 indicates the limit field, and 3 indicates the attribute field.

This counter is used by MMU commands that access multiple bytes within a descriptor. In general, the counter is handled automatically by the MMU commands. Only when a command could be interrupted—and intervening MMU commands issued—should this register be saved and later restored by the interrupting program.

### Status Registers

Six 8-bit registers contain information useful in recovering from memory trap conditions (Figure 9). The Violation Type Register describes the conditions that generated the segment trap. The Violation Segment Number and Offset Registers contain the segment number and upper byte of the segment address offset for the logical address that caused the segment trap. The Instruction Segment Number and Offset Registers contain the segment number and upper byte of the segment address offset for the last instruction before the segment trap was issued. The Bus Cycle Status Register records the status of the bus at the time the trap condition was detected.

Only violations caused by CPU access have trap information stored in the status registers; DMA violations cause Suppress to be asserted, but the Status Registers are not altered. Thus if a DMA violation occurs between a CPU violation and entry to the trap service routine, the service routine still has the trap information available to process the trap. It is the responsibility of the DMA device to save enough information in the event of a violation so that a software DMA violation service routine can process the violation correctly.

Eight flags in the Violation Type Register describe the cause of the segment trap. Four flags correspond to access protection modes in the segment descriptor attribute mode. A read-only violation sets the RDV flag, a system-only violation sets the SYSV flag, a CPU access to a CPU-Inhibit segment sets the CPUIV flag, an execute-only violation sets the EXCV flag.

Three flags correspond to addressing violations or warnings. The Segment Length Violation Flag (SLV) is set whenever the offset of the logical address falls outside the memory space allocated to the segment. The Primary Write Warning Flag (PWW) is set whenever a write occurs in the last 256 bytes of a segment whose Direction And Warning Flag is set (that is, for segments being used as stacks where the top of the stack is within 256 bytes of the allocated memory space of the segment). The Secondary Write Warning Flag (SWW) is similar to the PWW flag; only it is set when the CPU is in system mode, a stack push is being performed to a segment with a Direction And Warning Flag set, and some other addressing violation or warning has occurred (the EXCV, CPUIV, SLV, SYSV, RDV or PWW flags have been set). When the SWW flag is set it indicates that the system stack is in danger of overflowing its allotted memory. Once the SWW flag is set, further write warnings are suppressed.

This prevents the system from repeatedly being interrupted for the same warning while it is in the process of eliminating the cause of the warning.

The final violation-type register flag to be discussed is the Fatal Condition Flag (FATL). This flag is set when any other flag in the violation type register is set and either a violation is detected or a write-warning condition occurs in normal mode. This flag is not set during a stack push in system mode that results in a warning condition. This flag indicates that a memory access error has occurred in the trap processing routine. Once this flag has been set, no Trap Request signals are generated on subsequent violations. However, Suppress signals are generated on this and subsequent CPU violations until the FATL flag has been reset.

The Bus Cycle Status Register contains information pertaining to the status of the bus when a trap condition is detected. This includes CPU Status (ST0–ST3), plus flags indicating whether a read or a write was being performed and whether or not the N/S line was asserted.

The Violation Segment Number and Offset Registers record the first logical address to cause a trap. Only the high-order byte of the offset is saved, however, so that external support circuitry is needed to save the low-order eight bits of the logical address offset. If the trap occurred during the instruction fetch cycle, this information is the logical address of the instruction; otherwise it indicates the

```
<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 9. MMU Violation Information Registers
```
Status
Registers (Continued)

logical address of a data item which was to be accessed.

The Instruction Segment Number and Offset registers record the logical address of the last instruction fetch that occurred before the trap. Only the high-order byte of the offset is saved, however, so external support circuitry is needed to save the low-order eight bits of the offset.

If an instruction fetch caused the trap, these registers indicate the logical address of the previous instruction. Such information is useful if the preceding instruction was a branch instruction to an invalid address since—in this case—these registers indicate which branch instruction led to the erroneous situation. If a data reference caused the segment trap, then these registers indicate the logical address of the instruction that specified the illegal access.

Stack
Segments

Segments are specified by a base address and a range of legal offsets to this base address. On each access to a segment, the offset is checked against this range to insure that the access falls within the allowed range. If an access outside the segment is attempted, a Trap Request and a Suppress signal are generated.

Normally the legal range of offsets within a segment is from 0 to 256N + 255 bytes, where 0 ≤ N ≤ 255. (N is the value in the limit field of the segment descriptor.) However, a segment may be specified so that legal offsets range from 256N to 65,535 bytes, where 0 ≤ N ≤ 255. The latter type of segment is useful for stacks because the Z8001 stack-manipulation instructions cause stacks to grow toward lower memory locations. Thus, when a stack grows to the limit of its allocated segment, additional memory can be allocated on the correct end of the segment. As an aid in maintaining stacks, the MMU detects when a write is performed to the lowest allocated 256 bytes of these segments and generates a Trap Request. No Suppress signal is generated so the write is allowed to proceed. This write warning can then be used to indicate that more memory should be allocated to the segment.

The DIRW flag indicates that a segment is to be treated in this special way by the MMU. When the DIRW flag is set, the range of allowed offsets is from 256N to 65,535 bytes and writes into the range 256N to 256N + 255 generate Segment Trap but not Suppress, indicating a write warning.

Segment
Trap and
Acknowledge

The Z8010 MMU generates a Segment Trap whenever it detects an access violation or a write warning condition. In the case of an access violation, the MMU also activates Suppress. Suppress can be used to inhibit memory writes and to request that special data be returned on a read access. Segment Trap remains Low until a Trap Acknowledge signal is received. If a violation occurs, Suppress is asserted for that cycle and all subsequent CPU memory references until the end of the instruction. Intervening DMA cycles are not suppressed, however, unless they generate a violation. Violations detected during DMA cycles cause Suppress to be asserted during that cycle only; no segment trap requests are ever generated during DMA cycles. This is because the CPU would not be able to respond to these traps until the conclusion of the DMA cycle.

Segment traps to the Z8001 CPU are handled similarly to other types of interrupts. To service a segment trap, the CPU enters a segment trap acknowledge cycle. The acknowledge cycle is always preceded by an instruction fetch cycle that is aborted. The MMU has been designed so that this dummy instruction fetch cycle is ignored. During the acknowledge cycle, all enabled MMUs use the Address/Data lines to indicate their status. An MMU that has generated a Segment Trap request outputs a 1 on the A/D line associated with the number in its ID field. An MMU that has not generated a segment trap request outputs a 0 on its associated A/D line. A/D lines for which no MMU is associated remain 3-stated. During a segment trap acknowledge cycle, an MMU uses A/D line 8 + i if the content of its ID field is i.

Following the acknowledge cycle, the CPU automatically pushes the program status words and program counter onto the system stack, and loads a new program status word and program counter from the program status area. The Segment Trap line is reset during the segment trap acknowledge cycle, and no Suppress signal is generated during the stack push. If the store creates a write warning condition, a segment trap request is generated and is serviced at the end of the context swap; the SWW flag is also set. Servicing this second Segment Trap request also creates a write warning condition, but—because the SWW flag is set—no Segment Trap request is generated. If a violation rather than a write warning condition occurs during the context swap, the FATL flag is set rather than the SWW flag. In this case, subsequent violations cause the Suppress to be asserted but not Trap Request. Without the SWW and FATL flags, trap processing routines that generate memory violations would repeatedly be interrupted and called to pro-
Trap and Acknowledge (Continued)

The CPU routine to process a trap request should first check the FATL flag to determine if a fatal system error has occurred. If not, the SWW flag should be checked to determine if more memory is required for the system stack. Finally, the trap itself should be processed and the violation type register reset.

<table>
<thead>
<tr>
<th>Commands to the MMU</th>
</tr>
</thead>
<tbody>
<tr>
<td>When a memory management system must read or change information in the MMU to respond to a segment trap or to re-organize the physical memory, it can issue control commands to the MMU. These commands fall into two generic categories: reset commands and read/write commands. Reset commands are simply orders to the MMU to set or clear specified fields. For these commands, the Z8001 Special I/O output command can be used with the destination field set to be the MMU command code corresponding to the desired action.</td>
</tr>
</tbody>
</table>

Read and write commands are slightly more complicated because they consist of both commands and data. Such commands to the MMU are issued using the Z8001 Special I/O instructions. These instructions have a source and a destination field. For an input instruction, the source field contains an MMU command code and the destination field indicates where in primary memory the data is placed. For an output instruction, the destination field contains an MMU command and the source field indicates where the data to be written into the MMU resides in memory.

The high-order byte of the command contains the opcode for that command; the low-order byte of the command can be used to specify the particular MMU to be accessed. The MMU does not receive information on AD0-AD7, so external circuitry must decode information on these lines during the Special I/O commands and then select a particular MMU. The encoding of the low-order byte is dependent upon the system implementation. This paper always uses the convention that bit 1 specifies MMU number 1.

The reset commands to the MMU are: Reset Violation Type Register, Reset SWW Flag In Violation Type Register, and Reset Fatal Flag In Violation Type Register. Resetting the Violation Type Register is similar to a hardware reset in that it clears this register and returns the internal control of the MMU to an initial state (as if no violation had occurred since system initialization). Resetting the SWW flag or the FATL flag in the Violation Type Register clears these flags.

Two other commands are similar to reset commands in that they have no data associated with them. These are Set All CPU-Inhibit Flags in the segment attribute fields and Set All DMA-Inhibit Flags in the segment attribute fields, both of which cause all segment descriptors in the MMU to have the CPU or DMAI flags set, respectively. These two set commands can be useful in initializing address translation tables or when swapping between tasks. For example, when swapping between tasks the Set All CPU Flags command automatically makes the previous task's segments inaccessible to the next task, unless the system explicitly initializes the segment attribute field in these segments.

As an example of using the Special Output instruction SOUT to control an MMU, consider resetting the fatal flag of MMU #1. The MMU command opcode for this is "%14" (% denotes hexadecimal). The assembler syntax for the SOUT instruction is "SOUT destination field, source field" so that the instruction to reset the fatal flag of MMU #1 is "SOUT %1402, R0." Specifying register 0 in this instruction is an arbitrary choice—the content of this register is placed on the A/D lines during the data phase of the SOUT instruction, but it is ignored by the MMU. The low-order byte of the command (the destination field of the instruction) encodes which MMU is to reset its fatal flag. The convention followed in this paper is that MMU 1 is specified by setting bit 1 in the low order byte of the command. (Bit 1 set is hex "%02.")

The rest of the MMU commands consist of both operation and data. The following internal registers can be read or written: the Mode Register, the Segment Address Register, the Descriptor Registers and the Descriptor Selection Counter Register. A Descriptor Register can be read or written as a whole, or selected subfields can be accessed. In addition, by using the auto-increment feature of the Segment Address Register, successive Descriptor Registers can be accessed, or a selected field within successive Descriptor Registers can be accessed. For example, one Special I/O command in block mode could read a number of segment attribute fields. This is useful in determining which segments have been modified.

As an example of using the Special Output instruction SOUT to write data into an MMU, consider writing the contents of Register 6 into the Mode Register of MMU #2. The opcode for this command is "%00" and so the command is "SOUT %0004, R6." Here the high-order byte of the destination field contains the opcode and the low-order byte has bit 2 set (hexadecimal 4 if 0100 in binary) indicating MMU #2.
Commands to the MMU (Continued)

Certain MMU internal registers can only be read—there is no corresponding write instruction. This is because these registers contain information relating to a detected violation and thus it is not necessary to be able to write into these registers. These registers are the Violation Type Register, the Violation Segment Number Register, the Violation Offset Register, the Instruction Segment Number Register, the Instruction Offset Register and the Violation Bus Status Register. Although the Violation Type Register cannot be written, it should be noted that it can be cleared and that two of its flags can be individually cleared: the SWW flag and the FATL flag.

Direct Memory Access

DMA operations may occur between Z8001 machine cycles and can be handled through the MMU. The MMU permits DMA in either the System or Normal Mode of operation. For each memory access, segment attributes are checked and—if a violation is detected—a Suppress signal is generated. Unlike a CPU violation, which automatically causes Suppress signals to be generated on subsequent memory accesses until the next instruction, DMA violations generate a Suppress only on a per-memory-access basis. The DMA device should note the Suppress signal and record sufficient information to enable the system to recover from the access violation. No Segment Trap Request is ever generated during DMA (hence warning conditions are not signaled). There are no trap requests because the CPU would not acknowledge the request until the end of the DMA cycle.

Hardware and Software Reset

The MMU can be reset by either hardware or software mechanisms but note that they have different effects. A hardware reset occurs on the falling edge of the Reset input; a software reset is performed by an MMU command. A hardware reset clears the Mode Register, Violation Type Register and Descriptor Selection Counter. If the Chip Select line is Low while Reset is Low the Master Enable Flag in the Mode Register is set to 1. All other registers are undefined. After reset, the A/D and A lines are 3-stated. The SUP and SEGT open-drain outputs are not driven. If the Master Enable Flag is not set during reset, the MMU does not respond to subsequent addresses on its A/D lines. To enable an MMU after a hardware reset, an MMU command must be used in conjunction with Chip Select.

A software reset occurs when the Reset Violation Type Register command is issued. This command clears the Violation Type Register and returns the MMU to its initial state as if no violations or warnings had occurred.

Multiple-MMU Configurations

Z8010 MMU architecture supports system configurations that use more than one MMU. Multiple MMU devices can be used either to manage 128 CPU segments rather than the 64 supported by one MMU, or to manage multiple translation tables.

The Z8001 CPU generates logical addresses that can specify up to 128 different segment names. Because the MMU contains only 64 Segment Descriptor Registers, two MMUs are needed to perform address translation for 128 logical segments. Systems designed with only one MMU device still have the power and flexibility offered by memory management, although tasks in such a system are restricted to manipulating only 64 logical segment names. These names must either be 0 through 63 or through 127. If the MMU in a single-MMU configuration is set to translate segment names in one range and the CPU generates a logical segment name in the other range, the MMU does not perform address translation and no physical memory location is output. In this case, no request is made to memory. Therefore, a single-MMU configuration should have additional external logic to detect erroneous segment names and generate a Segment Trap and Suppress signal.

The Upper Range Select flag (URS) is used in multiple MMU configurations to indicate which group of logical segment names...
Multiple-MMU Configurations (Continued)

are to be translated by an MMU. When this flag is set, the Segment Descriptor Registers in the MMU are used in translating logical addresses in the range 64 through 127. When the flag is clear, the range is 0 through 63. Thus the URS flag corresponds to the most significant bit (bit 6) in the logical segment names that the MMU translates. Because this flag is under program control, the range of logical segment names can be changed during execution in System Mode.

MMU architecture also supports multiple segment translation tables. This feature is useful when separate tables are maintained for different tasks. Each task has its own table and switching between tasks requires enabling the appropriate MMU devices. In contrast, systems with only one translation table must either restrict the logical segment names that an individual task can use, or change the Descriptor Register entries whenever tasks are swapped. Two flags in the Mode Register, together with the N/S signal, are used in multiple table configurations.

The Multiple Segment Table (MST) flag indicates whether the configuration is being used to support multiple tables. When this flag is set, the MMU will compare the N/S line against the Normal Mode Select Flag (NMS) before generating a physical memory location on its Address lines. When the line and the flag match (both asserted or both de-asserted), the MMU is enabled and an address translation is performed (assuming the URS flag matches the most significant bit in the logical segment name). If the N/S line fails to match the state of the NMS flag, no translated address is generated by the MMU. The MST flag and the NMS flag are under program control and can be changed in System Mode.

The simplest multiple translation table configuration has one table for Normal Mode access and one for System Mode access. In such a configuration, the Multiple Table Flag is set in all MMUs and the N/S line of each MMU receives its input from the N/S output of the 28001 CPU. MMUs containing descriptors of system segments have the NMS flag clear, and those containing descriptors to be used in Normal Mode have the flag set. When the 28001 is in System Mode, the N/S line is Low and it matches the NMS flag in those MMUs whose Descriptor Registers contain system segment information. Therefore, these MMUs are used in address translation for system references.

When the 28001 is in Normal Mode, the N/S line is High and it matches the NMS flag in those MMUs whose Descriptor Registers contain user segment information. Consequently, these MMUs are used in address translation for user segments. In this configuration, system segments are separated from user segments. When the 28001 changes from Normal to System Mode of operation, the appropriate translation table is automatically selected. A more elaborate example of a configuration with multiple translation tables is given in the next section.

Examples

This section describes two 28001-Z8010 configurations: one contains two MMUs and one address translation table; the other contains seven MMUs and four address translation tables. These examples are given in sufficient detail to illustrate some of the major ideas in constructing memory-management systems around the Z8010 MMU. High-level block diagrams illustrate some of the major features of typical hardware configurations and short programs illustrate software techniques for using the MMU.

The first example system is the two-MMU configuration illustrated in Figure 10. The two MMUs are called MMU #1 and #2, and they are selected during a command cycle by AD1 and AD2 being Low, respectively. Since a Special I/O instruction is being used bit 0 must always be zero. Thus, when a low-order byte of a command is "%02," MMU #1 responds; when it is "%04," MMU #2 responds; and when it is "%06," both MMUs respond. (Note that AD1 is inverted before attachment to the CS pin.)

The A/D1 line, which controls MMU #1 through the Chip Select input, is first combined with the Reset line. This allows the Master Enable Flag to be set upon system initialization, so the logical addresses generated by the CPU are passed to the physical memory. This is done because—upon reset—the mode register is otherwise cleared, the Translate Flag is clear and addresses pass through the MMUs untranslated. The bootstrap program can therefore reside in absolute memory locations in the physical memory. If the Reset line is not an input to the Chip Select line, the Master Enable Flag would not be set during system initialization and the CPU would not be able to address memory through the MMUs.

Note that there is a direct path from the CPU and DMA to the system bus. This path is used during I/O and memory refresh because the MMUs are quiescent during these cycles. It is also used for data on memory reads and writes. Also, note that the Suppress line goes both to the memory, where it can be used to protect the memory from erroneous
writes, and back to the DMA device to save information upon the event of a DMA access error.

Of further interest in the example, address latches are used to buffer addresses between the Z8001 and a demultiplexed bus. This is required to demultiplex the address and data onto the bus. The address latch for AD₉-AD₁₅ may not be needed if the I/O device does not use separate address and data lines.

A detailed example indicates how such a system could be used. First, consider setting Segment Descriptor Register 65 to point to a read-only segment of 768 bytes starting at memory location %115200. The segment is to be accessed in Normal Mode. The Descriptor Register should be %115202 01. The first two bytes, %1152, indicate the starting location of the segment (note that the low-order byte of the memory address is all zeros and is not stored in the Descriptor Register). The third byte, %02, indicates that three blocks of 256 bytes have been allocated to this segment. The fourth byte, %01, indicates that only the read-only segment flag has been set.

To write this descriptor into the MMU, a copy of the descriptor should be created in primary memory and a Special I/O block transfer instruction used. The SOTIRB instruction can be used for this.

This instruction has the assembler syntax "SOTIRB destination, source, count register" where both the destination and source are registers. The destination register contains the command to the MMU, the memory location pointed to by the source register contains the first byte of the data to be transferred, and the Count Register contains the number of bytes to be transferred.

The opcode to load the Descriptor Register is "%0B". Segment Descriptor Register 65 is Segment Descriptor Register 1 of MMU #2, so the MMU command is "%0B04".

To specify which Segment Descriptor Register to write, it is necessary to load the Segment Address Register of MMU #2 with 1. The MMU opcode to do this is "%01" and so the command is "%0104." The segment number (in this case 65) is a parameter to the example routine, passed in register 0. The

Figure 10. A Dual-MMU Configuration
<table>
<thead>
<tr>
<th>Examples (Continued)</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT</td>
<td>R0, #6</td>
<td>!Test to see if Descriptor Register is in MMU #1!</td>
<td></td>
</tr>
<tr>
<td>JR</td>
<td>Z, OVER</td>
<td>!or MMU #2!</td>
<td></td>
</tr>
<tr>
<td>SOUTB</td>
<td>%0104, RH0</td>
<td>!Set SAR in MMU #2!</td>
<td></td>
</tr>
<tr>
<td>LD</td>
<td>R1, #%0B04</td>
<td>!Prepare to write descriptor!</td>
<td></td>
</tr>
<tr>
<td>JR</td>
<td>NEXT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OVER:</td>
<td>SOUTB</td>
<td>%0102, RH0</td>
<td>!Set SAR in MMU #1!</td>
</tr>
<tr>
<td></td>
<td>LD</td>
<td>R1, #%0B02</td>
<td>!Prepare to write descriptor!</td>
</tr>
<tr>
<td>NEXT:</td>
<td>LD</td>
<td>R0, #4</td>
<td>!Load count field–4 bytes!</td>
</tr>
<tr>
<td></td>
<td>SOTIRB</td>
<td>@R1, @RR2, R0</td>
<td>!Write descriptor!</td>
</tr>
</tbody>
</table>

descriptor to be written is another parameter to this routine: RR2 contains the address in memory where this information resides. The SOUTB instruction has a similar syntax to the SOTIRB instruction explained previously except that it writes one byte instead of a series of bytes, and the destination I/O address is in the instruction itself instead of in a register specified by the instruction.

The routine on this page initializes the Segment Descriptor. Its parameters are found in Register R0, which contains the segment number to be written, and in Register RR2, which points to the descriptor information in primary memory. Registers R0 through R3 are used by this routine.

Now suppose that the user tries to write into location &lt;&lt;65&gt;&gt; %9328. This causes a segment trap both because of the write to a read-only segment and because the access exceeds the segment limit. At the end of the instruction that has the illegal memory access, the CPU acknowledges the trap. During the trap acknowledge cycle, MMU #2 asserts AD10 (assuming its ID field is "010") and this information is placed on the system stack for the trap-handling routine.

The trap-handling routine reads the violation information registers from the MMU. The violation type register contains "%05" indicating both a length violation and a read-only violation. The Violation Bus Status Normal Register contains "%28". The first nibble indicates a write in Normal Mode was in progress and the second nibble indicates a memory data access cycle was in progress. The violation segment register contains "%41" indicating segment 1 of MMU #2 caused the violation (which is segment number 65), and the violation offset register contains "%93" indicating the high-order byte of the logical address offset. The operating system can then issue an error message to the user indicating a read-only violation to segment 65. Using the program counter that was stacked when the segment trap was acknowledged, the system can also indicate the next instruction that was to be executed. Note that in this system the low-order byte of the violation offset is lost. This condition is corrected in the next example system.
Figure 11 gives a high-level diagram of the second system to be discussed. This configuration contains 16 MMUs, and the A/D lines select the appropriate MMU when in Command mode. The major innovation in this example, aside from the additional MMUs, is the latch that retains the least significant byte of an address offset when a violation is detected. This latch is enabled when a segment trap is generated by an MMU and holds the low-order byte of the address that generates an access violation.

In addition, external decoding logic for selecting one MMU Chip Select line is included. Seven MMUs is the limit in one configuration without additional decoding logic for selecting one MMU Chip Select line. (The reason why AD0 cannot be used to control an eighth MMU is due to the Special I/O input convention of the CPU. When the CPU inputs a byte of information and AD0 is asserted, the data is taken from AD0-AD7, which are not driven by the MMU.)

Switching Tables in a 16-MMU System.
The 16-MMU configuration can support a memory management system designed with two MMUs permanently allocated to the operating system and the others allocated in pairs to different user tasks. Thus, seven user tasks can have translation tables resident in the 14 user MMUs, and switching between active tasks requires the appropriate MMUs to be enabled and disabled. This selection process can be effected by manipulating the Master Enable (MSEN) flags in the mode registers of the appropriate MMUs.
The routine performs the selective enabling of MMUs required by a task swap. This routine disables all user MMUs (thus disabling the currently enabled user MMUs), then enables the appropriate pair. (The system pair is always enabled.) The code selecting the new task is passed in register R1; it contains n, if task n is to be dispatched.

Two peculiarities of this example are worth noting. First, each user ID number corresponds to seven MMUs (for example, all upper-range user MMUs). The Segment Trap processing routine has to take this into account. Second, the Chip Select code is assumed to be as follows:

It is also assumed that %F8 will select all user MMUs.

```
CLR R0  !Clear R0!
SOUT %00F8,R0  !Disable all user MMUs by clearing their mode registers!
SLA R1,#1  !Multiply R1 by 2—the number of bytes in a memory word!
LD R1,TABLE(R1)  !Get the command word (opcode always %00) for user n, URS=0!
LDA RR2,DATA  !Get the new mode register bit pattern (%DA)!
SOUTIB @R1,@RR2,R0  !Send %DA to lower-range MMU and increment RR2 to DATA +1!
INC R1,#8  !Command word for URS=1!
SOUTIB @R1,@RR2,R0  !Send %FB to upper range MMU!
```

```
DATA: BYTES(%DA,%FB)  !Mode register bit patterns!
TABLE: WORDS (%8,%18,%28,%38,%48,%58,%68)
```

### Program to Switch Tables

<table>
<thead>
<tr>
<th>MMU Command</th>
<th>Opcode</th>
<th>Operation</th>
<th>Opcode</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summary</td>
<td>00</td>
<td>Read/Write Mode Register</td>
<td>0C</td>
<td>Read/Write Base Field And Increment SAR</td>
</tr>
<tr>
<td></td>
<td>01</td>
<td>Read/Write Segment Address Register</td>
<td>0D</td>
<td>Read/Write Limit Field And Increment SAR</td>
</tr>
<tr>
<td></td>
<td>02</td>
<td>Read Violation Type Register</td>
<td>0E</td>
<td>Read/Write Attribute Field And Increment SAR</td>
</tr>
<tr>
<td></td>
<td>03</td>
<td>Read Violation Segment Number</td>
<td>0F</td>
<td>Read/Write Descriptor And Increment SAR</td>
</tr>
<tr>
<td></td>
<td>04</td>
<td>Read Violation Offset (high byte)</td>
<td>08</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>05</td>
<td>Read Bus Cycle Status Register</td>
<td>10</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>06</td>
<td>Read Instruction Segment Number</td>
<td>11</td>
<td>Reset Violation Type Register</td>
</tr>
<tr>
<td></td>
<td>07</td>
<td>Read Instruction Offset (high byte)</td>
<td>12</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td>08</td>
<td>Read/Write Base Field In Descriptor</td>
<td>13</td>
<td>Reset SWW Flag In VTR</td>
</tr>
<tr>
<td></td>
<td>09</td>
<td>Read/Write Limit Field In Descriptor</td>
<td>14</td>
<td>Reset FATL Flag In VTR</td>
</tr>
<tr>
<td></td>
<td>0A</td>
<td>Read/Write Attribute Field In Descriptor</td>
<td>15</td>
<td>Set All CPU-Inhibit Flags</td>
</tr>
<tr>
<td></td>
<td>0B</td>
<td>Read/Write Descriptor (all fields)</td>
<td>16</td>
<td>Set All DMA-Inhibit Flags</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>17-1F</td>
<td>Reserved</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>20</td>
<td>Read/Write Descriptor Selector Counter Register</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>21-3F</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

00-2049-A  3-47
An introduction to memory management

Once used only on the largest computer systems, memory-management techniques will soon be used on a variety of high-level microprocessor-based systems

by D. Stevenson

The declining cost per bit of memory has led to systems with even larger memories, and the declining cost of logic has led to more powerful processors. Together, these two trends promote the sophisticated use of large memories, based on techniques commonly referred to as memory management. Automated memory-management systems date back to the Atlas computer project at Manchester University in the late 1950s. During the 1960s the concept was exploited in a number of timesharing machines (e.g. the Scientific Data Systems 940, General Electric 645, Digital Equipment Co. PDP-10), and during the 1970s was highly publicised in its manifestation as virtual memory (IBM 370). Until fairly recently, memory management has been associated only with large mainframe computers, but with the 1978 introduction of Digital Equipment's VAX 11 'super min', the concept has invaded the minicomputer market. Now, with the advent of single-chip memory-management units such as that available with the Zilog Z8000 processor, the concept is about to arrive in microprocessor-based systems.

Memory management has two functions: the efficient allocation and reallocation of memory space to executing tasks so as to optimise overall memory usage; and the protection of memory contents from unintended or unauthorised accesses by executing tasks. To keep overall memory usage optimised as demands on memory constantly change, dynamic relocation of tasks during their execution may be necessary, and this is accomplished by an address-translation mechanism. The restriction of memory access to prevent unintended or unauthorised accesses is accomplished by memory- attribute checking. Both operations occur with each memory access made during the execution of a program, and both are transparent to the user.

Address translation simply means treating the memory addresses generated by the program as logical or virtual addresses to be translated into actual physical-memory addresses before dispatching the memory-access requests to the memory unit. Memory-attribute checking means that each area of memory has associated with it information as to which tasks can access it and what types of access can be made by each task. Each memory reference is checked to ensure that the task has the right to access that location in the given fashion (for example, to read the contents of the location or to write data to that location).

Instead of a conventional linear address space, more elaborate memory-management systems simulate a hierarchical memory structure in which the memory consists of a collection of distinct memory areas, called segments. Access to this structured memory requires the specification of a segment and of an offset within that segment. Thus, instead of specifying, say, memory location 1050 in a linear address space, a task might specify memory location 5 in segment number 23. The actual location of the segment in the physical memory does not concern the task — the actual access is carried out via the address-translation mechanism, which is informed of the actual location of the segment by the operating software.

Generally, segments can be of variable size, within limits, and a user can specify the size of each segment to be used. Thus one user may be allocated

1 In a multiuser system, each user is aware of only those memory segments in his own 'personal' logical-memory space, and does not know where the segments are located in the system's physical memory
two segments, one of 2000 words for his Fortran program, and the other of 10,000 words for his data. Another user might be allocated three segments, of 3000, 6000 and 2000 words, respectively, for her Pascal program, data, and run-time stack. If the first user called his data segment 'segment 5', then the first word in his data set would be accessed by the logical address (5,0), indicating segment 5, offset 0. The memory-management system then translates this symbolic name into the correct physical-memory address.

Fig. 1 gives a conceptual realisation of these two users logical program spaces. The first user, user A, has his program segment called 'segment 6' and his data segment called 'segment 5'. The second user, user B, has her program segment called 'segment 5', her data segment called 'segment 12' and her stack segment called 'segment 2'. Notice that both users have named one of their segments 'segment 5', but they refer to different entities. This causes no problem since the system keeps the two memory areas separate. The situation is analogous to both users having an integer variable called 'I' in their programs; the system realises that these are two separate variables stored in different memory locations.

User A's data segment, 'segment 5', is 10,000 words long. If he tries to reference word 10,050 of segment 5, he gets an error message from the operating software indicating that he has exceeded the allocation limit for segment 5. Note that he does not accidentally access word 50 of segment 6; i.e. segments are logically distinct and unordered. A reference to one segment cannot inadvertently result in access to another segment. Thus, in this example,

A virtual end to microprocessor memory limits

All the 'super microprocessor' designs are based on the need to reduce software costs by facilitating programming, and one of the major causes of difficulty in programming any computer-like device is limitations on the size of available memory. By removing the 64 kbyte limit imposed by earlier devices, the 'super micros' have eased this problem, but experience with larger, usually mainframe, computers suggests that the user's programming task will be simplified even more when, in the near future, microprocessors become capable of supporting virtual-memory operation, which effectively removes all practical limits on memory size.

Virtual-memory operation works by using relatively inexpensive secondary memory, such as that provided by magnetic discs, to supplement the system's primary memory, which is necessarily built from relatively expensive random-access-memory components. However, since any program or item of data can only be accessed by the processor when held in the random-access primary memory, this can only be done by continuously 'swapping' blocks of program code or data between the two memories. This swapping is carried out automatically by the processor's hardware and operating software, so that the operation of the whole virtual-memory system is transparent to the user, who is aware only of having access to an extremely large personal memory space.

As an example of the use of virtual-memory operation is given by Digital Equipment's VAX-11 advanced-architecture version of its PDP-11 minicomputer. As a full 32-bit machine, the VAX-11 is capable of addressing over 4 \times 10^9 bytes of memory, and the virtual-memory system operates to effectively give each process, no matter how many are concurrently active, access to over 10^9 bytes of 'private' memory space. In practice, of course, not even the most demanding application requires any of its processes to have access to this huge amount of memory; thus the aim of removing all practical limitations on memory size has been achieved.

The VAX-11 processor implements its virtual-memory system by a relatively simple paging technique. The whole address space is divided into 512-byte 'pages' that can be swapped independently between primary and secondary memory. Each logical address used in the system is composed of a 23-bit page number and a 9-bit offset. At each attempted access,
user A is prevented from accidentally (or deliberately) accessing his program as though it were part of his data segment.

Fig. 2a illustrates one way that an operating software could arrange these segments in the physical memory. If demands on physical-memory space were to change, however, the operating software could dynamically relocate the segments (e.g. as shown in Fig. 2b), the relocation being completely transparent to the two tasks. In each case, the arrows indicate the address-translation or memory-mapping functions from the logical-address space of the users to the physical-memory locations allocated to them. The Figure also indicates the access attributes associated with each user's segments. For example, program segments are 'execute only' and data segments are 'read/write'. Thus a user is prevented from executing a data segment or writing into a code segment.

Fig. 3 illustrates what happens when both users have access to the same data set in primary memory, say the results of a questionnaire that both intend to analyse. Each user has a logical name associated with that data set to specify the segment in which the data set is to reside. Note that the two users have chosen to put the data set in different segments of their personal address spaces. The memory-mapping system translates these different segment names to the same physical memory locations. Thus user A's access to address (2,17) references the same physical memory location as user B's access to address (7,17). The shared data segment is marked 'read only' to prevent either user from deliberately or accidentally changing the data.

Before proceeding to the mechanism

the system's memory-management unit checks to see if the desired page is in the primary memory, and if so translates the logical address to the appropriate physical address. If the page is not in primary memory, it is 'swapped in' from the disc. All this is relatively straightforward — the complication comes in during the design of the 'paging algorithm' by which the operating software decides which page currently in primary memory can most readily be 'swapped out' to free space for the incoming page. The efficiency of the whole system depends critically on the choice of the correct swapping algorithm, and in computer-science terms this choice is 'non-trivial', or in other words extremely difficult.

With the forthcoming announcement of the National Semiconductor 16000 'super micro' range, virtual-memory operation of this kind will become feasible in microprocessor systems for the first time. The NS16082 memory-management unit (m.m.u.), which will act as a coprocessor to the NS16000 main processor, will support a paging system of virtual-memory operation rather like that used on the VAX-11.

Because of the NS16000's use of 24-bit addresses, each virtual-memory space will be initially limited to only 16 Mbytes, but later expansion should increase this substantially. The m.m.u. will provide fast associative storage for active address-translation tables within its internal memory, using a cache approach to ensure that only 5% of accesses require reference to the full translation tables stored in primary memory. On detecting that a required page is not in primary memory, it will send a 'abort' message to the main processor, which is equipped with a special hardware mechanism to 'roll back' its state to what it was at the start of the aborted instruction. National Semiconductor claims that, with these features, and with an appropriate operating system to control them, the design of a full virtual-memory system should not be significantly more difficult than the design of any other microprocessor-based system.

Will such virtual-memory microprocessor systems ever become widely used, however? One development that may make them extremely attractive is that of denser, less expensive, magnetic-bubble stores. A relatively inexpensive system could then be based on a 'super micro', a relatively small (say 128 kbyte) primary memory, and 1-2 Mbyte of fast non-volatile bubble storage. Such a system, occupying a single board, might well exhibit a performance approaching that of traditional mainframe systems.
Memory management comes to micros

A survey of recent product announcements reveals that the microprocessor manufacturers all agree on the importance of providing memory-management facilities for the next generation of microprocessor-based systems. All the new-generation 'super microprocessors' have been designed with the use of memory management in mind, and existing microprocessor families, such as the Texas Instruments 9900 range, are being extended by the provision of 'add-on' memory-management units. Although it might at first seem that the use of memory-management techniques could only be justified in specialised high-end applications, the low cost of the large-scale-integration hardware and packaged operating software now becoming available may soon make sophisticated management a common feature of even relatively modest micro-based systems.

With the exception of Intel, all the microprocessor manufacturers have decided to use the traditional minicomputer approach to providing memory-management facilities, which involves the use of separate memory-management units (m m u's) located between the processor and memory. Using this approach, the m m u accepts memory-access requests from the processor on its input lines, performs address translation and memory-attribute checking as desired, then sends appropriately modified access requirements to the memory via output lines.

Typical of such m m u's is the Z8010 unit designed to provide memory-management facilities to systems based on the Zilog Z8000 microprocessor. In operation, the m m u receives 25-bit logical addresses from the processor, these addresses consisting of a 7-bit segment number and a 16-bit offset. These logical addresses are then translated into 24-bit physical addresses by using the segment number to address a 64-line table held within the m m u, adding the 16-bit segment starting address thus retrieved to the top 8 bits of the logical-address offset, and concatenating the lower 8 bits of the offset with the result of this addition (see Figure). The m m u then sends this 24-bit physical address to memory to complete the access.

The use of a 7-bit segment number enables the Z8000 to divide its memory into a maximum of 128 separate logical segments. A second m m u is being designed for the first if more than 64 segments are actually going to be in use at any one time. These segments may be of variable length, up to the maximum of 64 kbytes imposed by the 16-bit size of the offset, and may be located freely within the overall 8 Mbyte memory, subject only to the restriction of the starting address being a multiple of 256 bytes, a restriction imposed by the 16-bit size of the stored segment-starting addresses. Relocation of segments is achieved by the processor changing the contents of the appropriate entries in the m m u's segment-address table, which is done by the m m u's special input/output instructions.

As well as providing for address translation, the m m u also checks the attributes of the addressed segment, which are stored, along with its starting address, in the internal 64-line table. One attribute it always checks is the length of the segment, ensuring that the logical address provided does in fact lie within the declared segment boundaries. Other attributes relate to the type of access allowed, e.g. execute-only, read-only and read/write, and to check that the access being attempted does not violate these memory-protection attributes, the m m u needs to know for what purpose the processor is trying to access the specified segment. This is determined by monitoring four status lines connected to the processor, which indicate, inter alia, whether the processor is trying to fetch an instruction from memory, to access data from memory, or to manipulate a memory-based stack. If the access is not allowed by any of these segment's attributes, the m m u interrupts the processor via a special 'segment trap' line.

A unique feature of the Z8000 design is that the use of the four processor-status lines could be used to divide the system's memory additionally into special-purpose areas each capable of holding only one type of data. Thus, completely separate memories could be provided for program instruction, data and stacks. The distinction between user and system operation could double this to a total of six separate memories, each of which could be 8 Mbytes in length. Whether any user would actually want to partition his system's memory in this rather inflexible way, however, remains to be seen.

The operation of the m m u, although based on fast hardware logic, inevitably results in memory access suffering a certain additional delay. In the Z8000 system, this delay is minimised by arranging for the

of memory management, it is instructive to review the advantages of using this form of segmented address translation and attribute-based memory protection. The first advantage is that it permits the dynamic allocation of memory during the execution of tasks, i.e. tasks can be located anywhere in memory, and can be relocated as desired while their execution is suspended. The address-translation mechanism provides this flexibility because the task deals exclusively with logical addresses, and hence is independent of the addresses of the physical-memory locations it accesses. Moving the task to different physical-memory locations requires that the address-translation function be modified to reflect the change in physical memory location, but the task's code need not be modified. Of course, this flexibility does incur the overheads involved in managing the various address-translation tables required by the operating software, but these are normally outweighed by the advantages.

The second advantage is that it allows the sharing of common memory areas by different tasks. This is accomplished by mapping different logical areas in different tasks to the same physical-memory locations.

The third advantage is that it provides protection against certain types of memory access. This is accomplished by associating access attributes with each logical segment, and by checking the type of access to see if each access is permitted.

The fourth advantage is that it detects obvious execution errors related to memory accessing. This can be accomplished by checking each access to a segment to see whether the address falls within the physical-memory area allocated to that segment. It could also include affixing a read/write attribute to data to prevent a task from trying to execute a data segment, and affixing an execute-only attribute to code segments to prevent a task from trying to read or write data to this segment. Additionally, if a segment is used to hold a stack, the system could issue a warning to a task when the stack approaches the allocated limit of the segment. The task could then request the operating software to allocate more memory to the stack before the stack overflows and creates a fatal error.

The final advantage of such memory-management systems is that they separate user functions from system functions. For processors that distinguish between a 'system' mode and a 'user' mode of operation, this goal can be accomplished by associating a system-only attribute with operating-system segments so users cannot directly access the operating software and its data tables.

As a final point, it should be noted how segmentation can be used to support the development and execution of large, complex programs and systems. The concept of segmentation corresponds to the concept of partitioning a large system into procedures and data structures, each procedure and data structure being contained within a separate segment. A task can then invoke a procedure or subtask, or access a
processor to put the 7-bit segment number on its output lines one cycle ahead of the rest of the address. This gives the mmu additional time to retrieve the appropriate segment-starting address and to check the appropriate segment attributes.

From this account of the mmu's action, it will be seen that the Z8000 processor handles 23-bit logical addresses directly, each logical address comprising a segment number and appropriate offset. These 23-bit addresses can be stored as 32-bit 'long words' in pairs of 16-bit registers or in adjacent 16-bit memory words, and can be manipulated by all the Z8000's built-in 'long word' operations. For more efficient manipulation of short (up to 256 byte) segments, shortened logical addresses consisting of 7-bit segment numbers and 8-bit offsets can also be used; these shortened addresses fitting within an ordinary 16-bit word.

Very similar memory-management facilities are said to be planned for Motorola's 68000 'super micro'. The Motorola device, however, uses 24-bit logical memories to give a larger 16 Megabyte addressing range.

DENNIS MORALEE

Virtual memory

With the memory-management systems considered so far, it has been assumed that the actual physical memory available is always large enough for all the users' logical-address space to be simultaneously mapped onto it. In fact, further advantages can result from making even this physical-memory space 'virtual', and from mapping it into a two-level memory space, part of which is held in a relatively small 'true' physical memory, and part of which is held on a secondary-memory device such as a magnetic disc (Fig.4).

In operation, this virtual-memory arrangement relies on an extension of the address-translation scheme considered above. If a given segment is not currently in physical memory, the address-translation table indicates the fact, and links to a routine forming part of the operating software and capable of fetching the segment from secondary memory when needed.

Whenever an access is made to a segment missing from physical memory, the instruction execution is held in abeyance until the segment can be brought into the physical memory, and then the instruction is allowed to proceed with the memory access. The address translation is then performed, access protection is checked, and the instruction proceeds as if the segment had been in the physical memory at the beginning of the instruction. Thus the technique of demand swapping, or segmented virtual memory, means that the segments will not in general reside in physical memory until a task actually tries to access it.

Another technique of virtual-memory management is paging, which is also a method of partitioning a user's logical-address space and mapping it onto a two-level physical memory. Essentially, a paging system divides the logical memory into fixed-sized blocks, called pages. Like segments, the individual pages can be located anywhere in the physical memory, and a translation mechanism maps logical addresses to physical locations. There are two differences between paging and segmenting a logical memory. First, pages are of fixed size whereas segments are of various sizes. Second, under paging, the logical memory is still linear, i.e. a task accesses memory using a single number, rather than a pair as in segmentation.

The major advantage of paging is in treating memory as blocks of fixed sizes, which simplifies allocating memory to users and deciding where to place the logical pages in physical memory. The major disadvantage of paging is the difficulty of assigning different protection attributes to different areas in a user address space, because a paged memory appears homogeneous to the user and the operating system. Paging can, however, be combined with segmentation to produce a memory-management system with the advantages of both pag-
addresses and how the protection attributes are checked for each access, thus programs may be shorter. Also, management contains both the segment name and segmented memory (for example, the segment name or the offset within the segment). The alternative sets aside special registers that contain some of this information, for example, the segment name or the address in physical memory where the segment resides.

The advantage of the latter approach lies in the fact that fewer bits are needed in an instruction to specify addresses. Thus programs may be shorter. Also, because there is reduced traffic between the memory and the processor for fetching shorter instructions, a program may be executed faster.

On the other hand, these special registers must be manipulated to access more segments than there are registers, and this manipulation adds to the number of instructions, the program size and the execution time. In practice, these can destroy the advantages described above. If the special registers contain physical memory locations, these must be protected from user access to maintain the integrity of the system, and changing segments requires system calls which can be time-consuming if too few registers are supplied.

In either case, address translation is performed by adding the logical-segment offset to the address of the physical-memory location where the segment begins. Thus, when an address of the form \((a,b)\) is presented to the translation mechanism, the segment name \('a'\) is used to determine where segment \('a'\) resides in memory. Assume that it resides in locations 10000 to 25000. Then the actual memory location \((a,b)\) is memory location 10000 + \(b\). The major option in implementing this type of address translation is in determining the segment's location in physical memory. When special registers have been set aside to contain the starting location of the segment instead of putting all address information in the instruction, the addressing mechanism is similar to using the segment register as an index register or a base register.

When logical addresses are either completely specified in the instruction or when the special register contains the segment's symbolic name rather than its physical-memory location, a table must be used to translate the segment's name into its physical-memory location. The table may have an associative capability, i.e., the segment name is presented to the table and it automatically returns the physical-memory location where the segment begins. Alternatively, the table could have one entry for every possible segment name, with the starting address of each segment in use stored as part of the table entry.

A number of other segment attributes can also be stored in the address-translation table and checked during each access. One of these is the allocated length of the segment, and each access is checked to see if it falls within the bounds of the segment.

Another type of attribute deals with ownership or class of ownership: tasks are grouped into classes, and only those in certain classes are permitted to own and therefore access a given segment. The simplest example is the 'system versus 'user' classification, where tasks are either one or the other, and which they are determines whether or not they can access a given segment.

Other types of attributes that can be associated with a segment involve modes of accessing, for example 'read-only', 'read/write' or 'execute-only'. Attributes can be either permissive or prohibitive; for example the 'write' attribute can mean 'writing to this segment is permitted' or 'writing to this segment is prohibited'.

A final issue in the mechanics of memory-management systems is the implementation of the protection attributes. These may be associated either with the logical-address space or with the physical memory itself. Associating access attributes with the logical segment permits a more versatile memory-management scheme because different users can access the same physical segment and have different access attributes.

In a virtual-memory system, the system's 'physical memory' is split between a relatively small random-access 'main' memory and a secondary-memory disc store. If a program attempts to access a segment not currently stored in main memory, the operating software retrieves it from the disc, and processing continues transparently to the user.
associated with their accessing.

Other information that can be associated with each segment is associated not with the protection mechanism, but with other functions of the memory-management system. This information generally relates to the history of the segment; for example, whether a segment has been modified while resident in primary memory. If it has not been modified, and the system temporarily requires the memory space for another segment, the memory can be freed immediately; otherwise, the updated version of the segment must be stored in secondary memory, and the primary memory is not available until the segment has been saved. Although not strictly necessary, such information can improve the performance of the overall memory-management system.

References

David Stevenson is with Zilog Inc., 10460 Bubb Road, Cupertino, Calif. 95014, USA
Z8000 vs. 68000
Concept Papers

Introduction

July 1981

Introduction

The Z8000 and 68000 are similar CPUs, but important differences exist between them. The following concept papers discuss several substantive differences that design engineers should consider when trying to choose between these two CPUs.

Both the Z8000 and 68000 are classified as 16-bit CPUs, although each offers many of the attributes of a 32-bit CPU; each was designed with provisions for compatible expansion to a full 32-bit architecture. Each of these CPUs has an address space two orders of magnitude larger than the largest 8-bit CPU. Each has 16 central registers designed for general use (see the concept paper on the differences in register architecture). Each has a powerful instruction set, powerful addressing modes, and great regularity in the association of instructions with addressing modes. Each has a protected "user" mode, privileged instructions, and separate system and user stack registers. Each has automatic vectoring of traps and interrupts, with CPU status saved on a stack.

Critical Issues

The following concept papers focus on a number of critical design issues. This section summarizes the issues discussed and lists the key criteria used in addressing the relative merits of the Z8000 and 68000 approaches.

Memory Addressing

There is a sharp contrast between the segmented addressing model of the Z8000 CPU and the purely linear addressing model of the 68000 CPU. In examining these approaches, the following desirable attributes for a memory addressing scheme should be recalled:

- An addressing model that mirrors program organization
- Provision for access protection
- Provision for memory mapping

• Support for dynamic relocation
• Support for sharing
• Support for stacks

I/O Addressing

The Z8000 CPU has separate address spaces for I/O and memory; the 68000 uses memory-mapped I/O. The designer evaluating an I/O addressing mechanism should consider:

- Naturalness of the programming model
- Protection of I/O references
- Complexity of external interfacing logic
- Potential for performance improvement
- Provision for the block I/O function

Address/Data Bus

The Z8000 and 68000 CPUs use asynchronous address/data bus protocols. The Z8000 time-multiplexes a single set of lines for addresses and data, whereas the 68000 uses separate lines for addresses and data. In choosing between these approaches, the designer must consider:

- Performance limitations
- Complexity of interface to peripherals chips
- Optimal use of CPU pins

Register Architecture

The Z8000 and 68000 CPUs are similar in their register architectures, but they differ in significant details. Points that should be considered are:

- General vs. special-purpose use of registers
- Availability of registers of all necessary sizes
- Addressability of subregisters
- Extensibility of the register set
Operating System Support

The Z8000 and 68000 both provide many architectural features designed to assist in implementing the "system" portions of large and small applications, but there is a difference in the degree to which this area was addressed in the two designs. The Z8000 designers gave careful consideration to a thorough, unified approach to operating system support. As a result, the Z8000 is much stronger in this area than the 68000. The discussion of this area covers all of the following architectural support features for operating systems:

- Restriction of access to CPU and memory
- Memory mapping
- Sharing of programs and data
- Program relocation
- Stacks
- Context switching
- I/O system and interrupts
- Distributed control
- Support for conventions
The Z8000 and the 68000 take quite different approaches to register architecture. The principal points of difference are:

- General purpose vs. special purpose registers
- Pairing vs. telescoping of subregisters
- Extensibility of the register sets

This difference in register architecture results in generally simpler programming of the Z8000 than of the 68000. Several aspects of this are:

- Information in a Z8000 register never has to be moved before being used as an address or in arithmetic operations.
- The Z8000 uses the same op codes for arguments in any of the registers. This is in contrast with the 68000's separate op codes (e.g., ADD and ADDA for operations on the two register sets).
- The Z8000 uses the same addressing modes for all of the registers. This is in contrast with separate 68000 addressing modes like "data register direct" and "address register direct."

The net effect of these differences is that with regard to register handling the job of the compiler writer is easier with the Z8000 than with the 68000 and that compiled code for the Z8000 is likely to be more efficient than code for the 68000.

**PAIRING VS. TELESCOPING OF SUBREGISTERS**

The Z8000 instructions refer to byte registers, 16-bit registers, 32-bit registers and (occasionally) 64-bit registers. The 68000 refers to 16-bit and 32-bit address registers and to 8-bit, 16-bit and 32-bit data registers. On both machines, every register, except for those of the largest size, is contained in a register of the next larger size. Thus, every byte register is contained within a 32-bit register, and so on. On the 68000, this is a one-to-one relationship. Each 32-bit register contains exactly one 16-bit register, each 16-bit data register contains exactly one byte register. In each case, the subregister is the rightmost half of the larger register.
Z8000 Register Hierarchy

On the Z8000 a different scheme is used. The 16-byte registers are packed into 8 16-bit registers. The other eight 16-bit registers contain no byte registers. Similarly, the sixteen 16-bit registers are packed into the eight 32-bit registers, and the eight 32-bit registers are packed into four 64-bit registers.

The Z8000 arrangement facilitates the type-conversion operations that occur in higher level languages, since, for example, an 8-bit value stored in the rightmost eight bits of data register zero and sign extended to the whole 32 bits can then be referred to as the 32-bit R0, the 16-bit R0 or the 8-bit R0. On the Z8000, a similar situation is possible, but the names would be RR0, R1, R1. The price that is paid for this one feature is that the 68000 register hierarchy is inconvenient to use, while the Z8000 register hierarchy is a great programming convenience. For example, a Z8000 programmer can allocate four byte registers inside of one 32-bit register. On the 68000 a programmer would have to tie up four 32-bit registers to store the same four byte quantities. That's half of the data register set to do what can be done on the Z8000 in one eighth of the general purpose register set. If the Z8000 programmer wishes to use 16 byte registers, this can be done using only half of the register set. On the 68000, the maximum number of byte registers available is eight, and this ties up the entire data register set.

Another advantage of the Z8000 register hierarchy is that half of the 16-bit registers and all of the 32-bit and 64-bit registers have addressable halves. Half of the 32-bit registers and all of the 64-bit registers have addressable quarters.

The availability of this feature facilitates many programming tasks. On the 68000, there is no way to address the left half or any of the three leftmost quarters of any register. Such operations must be simulated with shift or rotate instructions.

EXTENSIBILITY OF THE REGISTER SET

The Z8000 instruction encoding uses 4-bit fields to designate registers; the 68000 uses 3-bit fields. This means that with no change in op codes and no change in instruction format, the Z8000 architecture will accommodate expansion of the general purpose register set to include 16 of each size of register. This means that eight 32-bit registers and twelve 64-bit registers can be added to the register set. The use of 3-bit fields and the telescoping of subregisters on the 68000 preclude a compatible extension of the number of registers of any given size and make introduction of 64-bit or larger registers extremely wasteful of register space.

SUMMARY

The Z8000 and 68000 register architectures are similar, but there are important differences. The 68000 uses special purpose address and data registers, the Z8000 uses a general purpose register file. The Z8000 uses pairing of smaller sized registers to make larger sized registers, the 68000 telescopes subregisters into the rightmost portions of larger registers. The Z8000 provides for compatible enlargement of the register file, the 68000 does not. In each case, the Z8000 approach is seen to be superior.
The Z8000 and 68000 take very different approaches to the addressing of I/O transactions. In the 68000, I/O addresses and memory addresses share the same address range. This is called memory-mapped I/O. References to I/O addresses are made exactly like references to memory addresses, using the same instructions and addressing modes. The processor does not know, when it engages in a read or write, whether it is talking to memory or to an I/O device.

In the Z8000, there is a separate address range for I/O transactions, and separate instructions are used. The processor always knows which kind of transaction is being conducted. The same physical address/data lines are used for the two kinds of reference; the status lines ST₃-ST₀ distinguish between them.

Several advantages have been claimed for memory-mapped I/O:

- Regularity - the same instructions and addressing modes are available for I/O as for memory.
- Simplicity - the size of the instruction set is reduced, since there are no I/O instructions.
- Ease of implementation - there is no need to design separate I/O bus protocols.

As to regularity, the kinds of operation performed on I/O ports are limited, as are the kinds of addressing that are useful in I/O operations. Furthermore, there are special needs of I/O operations that are different from those of memory operations (e.g., block transfers to a fixed address).

The 68000 design recognizes the fallacy of the regularity argument by introducing the MOVEP instruction—a block transfer of the bytes of a word or longword to consecutive even-addressed or consecutive odd-addressed bytes of memory. No 68000 instruction is provided for block transfers to a fixed address in memory.

The MOVEP instruction and the missing block I/O instruction also demolish the simplicity argument. Separate instructions are necessary because the two kinds of operation are different, and if the separation is not made explicit, an additional instruction will be necessary, as was done on the 68000.

In regard to the ease of implementation argument, I/O and memory transactions on the Z-Bus are only trivially different (I/O has an added cycle). The difference between the Z8000 and the 68000 bus protocols is not in ease of implementation. The difference is that the 68000 is locked into a single bus, while the Z8000 has the potential for future separation to improve performance.

Upon closer inspection, memory-mapped I/O has, in fact, many disadvantages.

- It makes protection of I/O references impossible at the instruction level— I/O instructions can't be privileged, because there are no I/O instructions.
- It creates "holes" in the memory address space, so that certain addresses—possibly localized, but potentially anywhere—cannot be used for memory addresses by any program.
- It prevents a compatible separation of I/O and memory buses—blocking an important path to performance improvement.

The question of protection is important in the design of operating systems. The I/O function is usually controlled by the system and prohibited to users, so it makes sense to make I/O instructions privileged. On the 68000, there are no I/O instructions (except for MOVEP, which is not privileged), so I/O instructions cannot be privileged. The only way to achieve this kind of protection on the 68000 is to assign to an external device the job of recognizing I/O addresses and preventing access to these addresses when the processor is executing in user mode.
The problem of "holes" in the memory address space can be partially alleviated by placing I/O addresses at one end or the other. (The 68000 sign extension of "short" addresses encourages this.) Nonetheless, the addresses are missing from the memory address range, and a runaway program could inadvertently store into these addresses, causing unpredictable results, including writing to tape or disk.

Finally, since no specific areas of the memory address range have been pre-assigned to I/O, the CPU has no way of knowing whether the transaction it is conducting is for I/O or memory. As a result, the potential performance improvement arising from separation of the I/O and memory buses is forever unavailable to the 68000. On the other hand, in keeping with the philosophy of "economy of means"—a major Z8000 design criterion—the Z8000 offers both the economy of using one bus for both I/O and memory and the potential for future separation.

In summary, the Z8000 design, by recognizing the distinction between I/O and memory operations, has achieved the following advantages over the 68000 I/O architecture:

- A natural programming model that easily incorporates the important block I/O function and avoids awkward instructions like the 68000's MOVEP.
- Protection—through making I/O instructions privileged and through having a separate I/O address space that no wild memory access can reach.
- Potential for future performance improvement through the separation of I/O and memory buses or through different handling of I/O and memory transactions, even on the same bus.
The Address/Data Bus

Z8000 versus 68000

Concept Paper

October 1980

The Z8000 and 68000 address/data buses are similar in that both use asynchronous protocols. They differ in that the Z8000 time multiplexes one set of lines for address and data, while the 68000 uses two separate sets of lines. The trade-off involves the higher potential performance of separate, dedicated lines versus the more effective use of the limited numbers of pins available for chip packages.

Dedicated lines have a potential for improved performance when one device has access to both the address and the data and can send them out simultaneously. The principal occurrence of this situation is a write to memory. There are several reasons why this potential advantage is of little consequence in a comparison of the Z8000 and the 68000.

- Reads from memory (including instruction fetches) occur roughly eight times as often as writes. A read from memory does not benefit from the separation of lines, since the address must be sent from the CPU to the memory before the memory can retrieve the data or instruction in question and send it back to the CPU.
- In the case of writes to memory, most memory chips are incapable of simultaneously accepting both the address and the data to be stored.
- Even with a memory chip that is capable of accepting addresses and data simultaneously, the 68000 still achieves no performance benefit, since 68000 write instructions are two cycles longer than read instructions (six cycles for writes vs. four cycles for reads) in order to allow the data bus to be turned around at the beginning and at the end of each write.

Considering the other side of this trade-off, the use of separate address and data lines results in the need for 16 (and, in the future, 32) pins that could be utilized to greater advantage. Looking just at the CPU, the 68000 faces all of the price, power and reliability problems of a 64-pin chip with no more capabilities than are provided by the Z8000's 48-pin package. When improved manufacturing technology allows economical and reliable expansion of the Z8000 to a 64-pin package, the 16 additional pins will provide greatly increased capabilities.

In addition to the more effective use of CPU pins, the multiplexing of address and data lines provides a means of addressing directly the internal registers of peripheral chips without the need to dedicate pins of the peripheral chip to separate address lines. Since at least eight data lines must generally go to a peripheral chip, these can be used during the addressing phase of an instruction to address a chip's internal registers (with the remaining eight I/O address lines possibly being decoded by external chip-select logic). This simplifies the programming of and access to peripheral chips by eliminating the separate address setup cycle required by an unmultiplexed peripheral interface.

In summary, the use of separate address and data lines gains little in performance, especially on the 68000 with its extra-long memory write instructions. It is wasteful of hard-to-come-by CPU pins and encourages a cumbersome interface for addressing peripheral chips.
Z8000 vs. 68000
Segmented vs. Linear Addressing

Concept Paper

November 1980
INTRODUCTION

The Z8000 and the 68000 use fundamentally different models for memory addressing. The Z8000 uses segmented addressing. The 68000 uses linear addressing. We shall define these terms and explain why segmented addressing is a superior method.

Segmented addressing is a "higher-level language" for memory addressing. That is, it is a way for the programmer to think about and refer to the computer's memory in terms that are natural to programming rather than in terms of the memory's physical implementation. Linear addressing is the "machine language" of memory addressing. That is, with linear addressing, the programmer uses a model for the computer's memory that is very close to its actual hardware implementation. Before we state more specifically exactly what segmented addressing is and how it works, let's look at some of the memory addressing tasks that programmers face and see what kind of addressing model these tasks suggest.

MEMORY ADDRESSING

The programmer is concerned with a variety of programs, data areas, stacks, etc. (for all of which the general term "objects" is used) and with the interactions among these objects. What we mean by this is partly a question of how fine-grained our picture is to be. For example, we could say that a programmer deals with two objects: the program and the data. At the other end of the scale, we could say that the programmer deals with a multitude of objects—listing separately each instruction and datum. Between these alternatives there can usually be found for each programming situation a set of largely separate but interrelated objects. For example, for a Chess-playing program, the objects might include:

- Chessboard display program
- Current position representation
- Legal move generation program
- Move evaluation program
- File of previously evaluated positions
- Handling routines for previous position file
- Program to study published games

This program might run under control of an operating system that was also divided into objects, including:

- Task scheduler
- Memory allocator
- Secondary storage interface routines
- Terminal interaction routines
- Process status table
- System stack
- User process status tables

The example could be refined and enlarged, but these are good examples of what we mean by the objects that the programmer must deal with.

The traditional approach to dealing with these objects is to allocate portions of the computer's memory to each of them. A relocating loader might pack the programs together end to end and then allocate the data areas (of fixed sizes) end to end in the portion of memory not occupied by the programs. Since the only addressing model available with the earliest computers was linear addressing, each of the objects would receive an address directly related to (usually the same as) the actual memory address at which it was stored. These addresses were all numbers in the range 0 to N-1, where N was the total number of memory locations available. Every program that referred to any of these objects had to do so using this address.

Figure 1. Traditional Approach to Memory Allocation

PROBLEMS WITH THE TRADITIONAL APPROACH

This approach always presented problems, and as systems grew larger the problems grew exponentially. We shall review these problems and look at some early solutions. The problems can be summarized under the following categories:

- Invalid accesses
- Difficulty of accommodating objects whose sizes vary—like stacks or lists
Invalid Accesses

The problem of invalid accesses occurs even in the smallest systems and on the smallest computers. In its basic form, the problem occurs when a program erroneously uses an address as if it belonged to one object when it actually belongs to another. For example, if an array is 1024 bytes long and a program erroneously refers to its 1025th byte, then the reference will actually be to the first byte of the object stored in memory immediately following the 1024-byte array. If the erroneous access is a store operation, then the object following the array in memory will have been damaged.

Identity of the array as arguments and returns a validated memory address that the program can use for fetching or storing. The routine might also handle the actual fetching or storing, accepting data to be stored as another argument or returning the data fetched. In either case, the routine would validate the access by using the array identity as a key to a set of array attributes, including the array's length and location in memory.

For the stack example, a similar envelope would be placed around pushes and pops. Rather than using the machine's push and pop instructions, the program would call subroutines for these operations. Naturally, this approach entails a large software overhead.

Another type of invalid access occurs in even the most elementary systems, but it presents an urgent problem when several programs or sets of data—not necessarily related to one another—share memory simultaneously. This problem concerns the restriction of a program's accesses to those portions of the memory containing its own subroutines and data or—even more difficult—to portions of memory containing data or subroutines that it shares with another program and to which it is allowed only certain kinds of access (such as "read only" or "execute only").

The software envelopes discussed above can be extended to accommodate shared access to data, but it is difficult to place such envelopes around program accesses. Furthermore, these envelopes are voluntary; that is, a programmer who wishes to avoid them can.
usually discover enough information to be able to make the accesses directly. For situations of this sort hardware solutions were introduced. One such solution was the use of limit registers. For example, the operating system might set registers that defined the limits of the program about to run to be locations 10000 through 19999. In this case, the program is free to make references of any sort so long as the address used lies within the given range. An attempt, for example, to call a subroutine at address 20000 results in a "trap," and control is returned to the operating system.

These examples are an indication of some of the ways in which the problem of invalid accesses can manifest itself, and they show how early system designers attempted to solve them. Shortly we shall see how segmentation provides a complete solution to this problem.

**Objects of Varying Sizes**

In our stack example above, we saw the kind of problem that can arise when an object varies in size. We showed how an envelope around pushes and pops can detect invalid accesses before they occur, but we are still faced with the problem of what to do about them. In the example given above, there was only one stack, and it didn't run out of memory until the entire memory of the computer was exhausted. However, if we had many stacks to manage, we might want to assign a small amount of memory to each and then expand those that were about to overflow. If all accesses to stacks are through the envelopes that surround the push and pop instruction, this is no problem. The stack can merely be "continued" elsewhere in memory, and the gap in the actual memory addresses between the last location of the original stack and the first location of the extension will be completely concealed from and irrelevant to the program using the stack. Unfortunately, the way that stacks are ordinarily used does not lend itself to this approach. Frequently a program is allocated a block of stack space, which it then accesses using based addressing. That is, the actual memory address of the first location of a block of stack space is kept in a register, and accesses into the block are made by adding an index (from a register or from an instruction) to the base addresses in the register. This common practice is incompatible with the existence of gaps in the set of addresses assigned to the stack.

The solution to this problem (before segmentation was invented) was to allocate a larger contiguous block of memory to the enlarged stack—either by moving the stack to another part of memory or by moving something else out of its way so that it could be expanded where it was. This approach has two inherent problems: the processing overhead to move objects around in memory and keep the unused memory all in one place and the "relocation" problem of changing all of the base addresses of blocks of stack space that the program has in registers or in storage. The second problem is almost insurmountable, except in the most elementary cases.

The problem of accommodating objects whose sizes vary has as a special case the problem of creating and deleting objects dynamically. This problem arises in the simplest single-user systems—for example, "Initialization" code might be abandoned after it is executed once and the space given to a large data array. As with our other examples, however, the difficulties mount rapidly as the system becomes more complex. In particular, because of the difficulty of "relocating" addresses, the moving of objects that would be necessary in order to keep the unused memory in one place is avoided. The unused memory soon

![Figure 4. Why Stacks Must be in One Piece](image-url)
comes to be scattered about in small pieces, and it becomes increasingly difficult to find contiguous blocks of sufficient size to accommodate newly created or expanded objects, even when the total amount of unused memory is sufficient. This problem is known as "fragmentation" of memory.

![Figure 5. Fragmentation](image)

Traditionally, there has really been no solution to this problem other than to leave management of the assigned memory to the user program. The user is provided with tools like "chaining" commands and overlay structures in certain systems, but by and large, the creation and deletion of objects is simply treated as part of the "algorithm" that the program implements. Soon we shall see how segmentation allows system control of this function.

**Relocation**

In discussing the expansion of stacks we alluded to the "relocation" problem that arose when a stack was moved: all of the pointers into it (the base register values for accesses to blocks of stack space) become invalid. This is a special case of the general problem of dynamic relocation. After the loader has established linkages among the parts of the program, it becomes almost impossible to move any of them. This is another problem that had to wait for a hardware solution. This solution has been provided at several levels.

Dynamic relocation, that is, relocation that occurs after the initial load of the program, requires a mechanism that allows actual addresses to be determined at run time. The first approach to this is provided by various kinds of based addressing. Based addressing of program references is usually provided by PC-relative addressing: calls, jumps and loads of program constants are specified using an offset that is added to the actual program counter value to obtain the memory address. Based addressing of data references is also made using offsets—to be added to a stack pointer or other address register. Relocation effected through based addressing is called "user-controlled" relocation, since the setting of the stack pointer or other address register is under control of the running program. A better approach from the standpoint of reliability is "system-controlled" relocation. This kind of relocation can be provided using memory mapping.

Memory mapping is, in its simplest form, a translation mechanism that converts the addresses used by the running program (which now become called logical addresses) into the actual memory addresses (now called physical addresses). With memory mapping, the program always uses a fixed set of addresses, and relocation is achieved by a change to the translation mechanism. A simple example of this is provided by a mechanism similar to based addressing. A value is set into a base register, and the translation mechanism consists of automatically adding that value to any address used in the program. (The difference between this and based addressing is that with based addressing there is an

![Figure 6. Memory Mapping with a Base Register](image)
This very simple form of memory mapping quickly evolved in two directions: paging and segmentation. Paging is a natural extension of the linear addressing model (although it can also be applied to the linear addresses used within segments). We won't say any more about paging or segmentation at this point.

Sharing

A natural outgrowth of memory mapping is a mechanism for the sharing of objects among otherwise independent processes. Given a mapping mechanism (more sophisticated than the simple base register mentioned above) that allows different blocks of logical addresses to be mapped independently of one another, a program or data area in physical memory can correspond to different logical addresses for different processes. Thus, the shared program or data can reside at a convenient location in the logical address space of each process, and the mapping mechanism will cause references from each process to be mapped by that process's mapping scheme into the given physical locations.

We shall say more about memory mapping in conjunction with our discussion of segmentation. At this point, we should simply note that system-controlled relocation and sharing through memory mapping alleviate one of the problems that tends to occur with user-controlled relocation and non-mapped sharing: fragmentation of the address space.

SOLUTIONS

We have now discussed the major problems with the use of the linear addressing model and have looked at some early attempts to solve them. Now we shall look at the abstract addressing model provided by segmentation, and we shall see how the 8000 CPU and memory management unit have been designed to work together to provide an implementation of this model that incorporates memory mapping and access protection. We shall show how this unified approach alleviates all five of the major problems with linear addressing that we stated earlier.

Segmentation

Segmentation is the organization of the address space into a collection of independent objects. As we noted earlier, in each programming situation there can usually be identified a set of largely separate but interrelated objects. The segmented addressing model assigns to each of these objects a "name" and a linear address space. The "name" is, of course, a binary number, but we call it a name to emphasize the fact that there is no relation between objects implied by a numerical relationship between their "names."

For example, in the example given above, the chessboard display program could be assigned object 1, the current position representation could be 2, the legal move generation program could be 3 and so forth. The address of any location within the chessboard display program would then consist of the name 1, and an address within that object's linear address space. If this program occupied 2048 bytes, for example, then the addresses within object 1 would be (1,0), (1,1) ... (1,2047). One of the attributes of object 1 would be a length of 2048 bytes, and the mechanism responsible for the interpretation of segmented addresses would be aware of this attribute and would cause an appropriate error indication if an address of the form (1,N) with N>2048 were ever used.

Now consider the case of the current position representation — object 2 in our example. Let's suppose that this representation takes the form of an array of 256 bytes. The addresses of these bytes would be (2,0), (2,1), ..., (2,255). One means of referring to items of this array involves the use of indexed addressing. The address of the item referred to would be specified by giving the array base address of (2,0) in one place—in the instruction or in a register—and an index (also called an offset) in a register. The index is simply a number to be added to the second component of the segmented address. Thus, if the Index were 17, then the item address would be (2,17). That is, the address manipulation cannot affect the object name portion of the address — only the linear address within the object is affected.

Similarly, returning to the display program (object 1 in our example), the mechanism responsible for address interpretation performs a similar computation for PC-relative addressing. If the program contains a branch to "current location - 24" or a call to "current location + 1264" for example, then the offset given in the instruction is applied to the second part of the address. If the call were made from location (1,562), then 1264 would be added to 562, and the final address would be (1,1826).

Preventing Invalid Accesses

These examples show how segmented addressing helps to alleviate our first major problem with linear addressing: invalid accesses. Suppose, for example, that we had made a programming error that caused us to address the current position representation array by using an index value of 257. With a linear
addressing scheme, this would result in a reference to the second byte of whatever object follows the current position representation array in memory. Thus, we might overwrite the second byte of the legal move generation program if that happened to follow the array.

With segmented addressing, the address computation would result in an address of \((2,257)\). The mechanism that interprets addresses would discover that this address is incompatible with the declared length of the array (256 bytes), so an appropriate error indication would be generated.

\[
\begin{array}{c}
\text{SEG 1} \\
\text{SEG 2} \\
\text{SEG 3}
\end{array}
\]

Figure 7. Attribute Checking for Segmented Addressing

Once the mechanism has been established for the checking of accesses against the declared size of an object, it is a small step to add the checking of other attributes of objects. The problems we mentioned earlier, such as protecting one process's data or programs from accesses by another process or allowing "read only" or "execute only" accesses to a section of data or program, can be solved by associating attributes with the objects in question and checking these attributes against properties of the access. A write into a "read-only" object, a user access to a "system-only" object and other such invalid accesses can be identified and prevented.

Sharing and System-Controlled Relocation

Since physical memories do not usually have a segmented organization, a segmented addressing scheme must include a plan for memory mapping. As noted earlier, memory mapping provides the means of dealing with two of our other major problems with linear addressing: the difficulties of implementing system-controlled relocation and of sharing objects among otherwise independent processes. We shall see shortly the specific details of how this is accomplished on the Z8000.

Avoiding Fragmentation

Our other two major problems had to do with the difficulty of creating, deleting, shrinking or expanding objects dynamically. We saw that these operations could lead to fragmentation in a linear memory space and that there were additional problems when stacks were involved. It is easy to see that segmentation provides solutions for these problems, but rather than discussing them abstractly, we shall now look at how segmentation has been implemented on the Z8000 and how the Z8000 and the MMU work together. Then we shall see concretely how all of the major problems of linear addressing have been solved by segmentation.

THE Z8001 AND THE MMU

The Z8000 has been designed with a built-in segmented addressing model. Included within the 32-bit addresses used by the Z8001 are two fields: the segment name field and the "offset." The offset is an address within the linear address space of the segment. It is called an offset because in the interpretation of segmented addresses, the offset is added to the physical memory address of the "base" of the segment to obtain the physical address of the element in question. For example, if segment 5 has a base address in...
If the physical memory of 1024 bytes contained within the segmented address space of 5K (5,26) is 1050, because 1024 + 26 = 1050.

The Z8001 is designed to work with an external circuit called a Memory Management Unit (MMU), which keeps track of the base addresses corresponding to the various segments and performs the computation of the actual physical addresses. This MMU can also associate a variety of attributes with each segment and can perform the corresponding access checking, generating an error interrupt (called a "segmentation trap") in the event of an invalid access.

Another feature of this implementation is that seven bits have been assigned to the segment name field and 16 bits have been assigned to the offset. This means that there are up to 128 segments, and each of them presents a linear address space of 64K bytes. Furthermore, the external MMU circuit is designed to translate only the uppermost eight bits of the offset; the low-order eight bits are passed directly to the physical memory untranslated. The practical effect of this is that a segment base address in the physical memory must be a multiple of 256 (i.e., its low-order eight bits must be zeros), and the size of a segment (one of the attributes that the MMU checks) must also be a multiple of 256 bytes.

Implementing Structures Whose Size Exceeds 64K Bytes

Let's look at the effect of these implementation details on programming the Z8000 and on the efficiency of its operation. The most obvious effect is that no object can exceed 64K bytes in size; that is, any data structure that exceeds this size must consist of more than one segment. This is a genuine problem, although it rarely occurs. Fortunately, it can be solved through the use of software with very little overhead. For example, if you are dealing with an array of size greater than 64K bytes then you cannot use

LD RL1,RR2(R3)

to access the byte with index kept in R3 of the array whose base is in RR2. Rather, you must use a sequence like

ADD RR2, RR4   laddr index to base!
ADDB RH2, RL2   laddr overflow to seg name!
CLRB RL2       lclear "unused" bits!
LD RL1, @RR2   to access the byte with index kept in RR4 of the array whose base is in RR2. What you are doing in this case is placing several segments "end-to-end" and treating the segment name like a number. This approach is similar to "paging."

Speed of Address Translation with the MMU

A more positive aspect of the implementation details has to do with the computational overhead of using an external circuit for address translation and attribute checking. Two facts enter into this:

- Since the segment name field is not involved in the address computations of indexed, based or relative addressing, this field can be output to the MMU one cycle earlier than the offset portion of the address, so that the MMU gets a one-cycle head start on the address translation.
- The low-order eight bits of the offset, which go directly to the memory untranslated, are the bits needed first by the memory, so that the memory also gets a small head start on the transaction.

The combination of these two factors results in the use of an external MMU circuit that entails very little time penalty in memory accesses.

The point made in the previous paragraph needs to be stressed: the true independence of the segment name field from the offset in all address computations means that off-chip memory mapping can be achieved with very little overhead. This is an architectural advantage of the Z8000 that leads to an economical implementation. This can be seen by looking at how a nonsegmented CPU might achieve memory management. Undoubtedly, the approach will be a form of paging. In a paged system, the uppermost bits of the linear address are treated like a segment name field after the address computation is complete. Until the computation is complete, these bits are treated like part of a monolithic linear address—they can be changed in the course of the computation. Thus, while a paging scheme allows memory mapping and attribute checking, it suffers from many of the problems of linear addressing, and it cannot achieve overlap of MMU and CPU computational time that is available with the Z8000 because of its true segmentation scheme. The only antidote to the computational overhead of an off-chip MMU for a linear addressed machine is to design an on-chip MMU, and with the current technology, that approach is likely to lead to a design that is short on features.

MMU Support for Stacks

One more point worth mentioning about the way that the Z8000/MMU combination implements segmented addressing concerns the use of stacks. Earlier, we noted some of the problems that are associated with dynamically expanding stack sizes. The most difficult of these problems concerned the correction of pointers into the stack when a stack was
moved to another location to accommodate a larger size. Naturally, this problem goes away with memory mapping, since the logical addresses of the locations already used on the stack don't change when it is physically relocated in memory. Furthermore, the MMU accepts as one of the attributes of a segment that it is to be used for a stack. This has two main consequences:

- There is a nonfatal stack warning interrupt that occurs when the stack is nearly full, i.e., when an access is made into the last 256 words allocated to the stack.
- The memory address computation and size specification method are altered to take account of the fact that stacks grow downward in memory from the highest addresses toward the lowest.

Just as there are some who argue that higher level languages are "inefficient" and that they don't allow the programmer the total flexibility of assembly language programming, there are also those who adamantly reject segmentation and cling to linear addressing. The truth is that there is merit to their argument. Just as higher level languages may be inappropriate for very small systems, so also may segmentation represent "overkill" in a small memory space. The Z8000 answer to this problem is to provide large enough segments that small applications can be implemented completely within the bounds of one segment. The Z8000 CPU is provided with a mode in which addresses consist only of offsets, so that no references occur outside of the 64K byte linear address space of one segment. In fact, for applications of that size, a smaller package is provided that does not have the eight pins dedicated to the segment name output and segment error interrupt input; this smaller version cannot enter the segmented mode of operation at all.

It is a matter for subjective judgment to decide where to draw the lines between systems that are too small for segmentation, systems in which segmentation is desirable but inessential, and systems that are so large that segmentation is mandatory. The Z8000 architecture provides for a 16-bit linear address space but demands segmentation for any size above 16 bits. In its 23-bit address space, it is possible that clever, well disciplined programmers could manage to handle unrestricted linear addressing. In its ultimate 32-bit address space, there is no doubt that segmentation is the only viable approach.

This concern for the future expansion to 32-bit address spaces greatly influenced the decision to use segmented addressing in the 23-bit version. The Z8000 represents a break from the architecture of the 800; it seems short-sighted to ask designers moving from 8-bit to 16-bit or 23-bit systems to face one architectural upheaval today and another in a few years. This is in contrast with the situation of designers who adopt the 68000 today and who will have to face another architectural upheaval if true segmentation is introduced—that occurrence seems inevitable if the address space increases in size to 32 bits.

CONCLUSION

This concludes our discussion of the specific details of the implementation of segmented addressing and memory mapping on the Z8000. We have discussed the many problems associated with linear addressing, the solution provided to these problems by the segmented addressing model and the details of segmentation on the Z8000. We have shown that segmented addressing is clearly superior to linear addressing.

Figure 9. Stack Segments
SUMMARY

The segmented addressing used by the Z8000 is a higher level language for memory addressing. This method is superior to linear addressing, the "machine language" of memory addressing, because it provides a model that is natural to programming.

The traditional approaches to memory allocation based upon the linear addressing model cannot solve the following problems:

- Invalid accesses
- Accommodating variable-sized objects
- Fragmentation arising from dynamic creation and deletion of objects
- Dynamic relocation
- Sharing

The Z8000/MMU combination provides a segmented addressing facility that solves the above problems and provides the following benefits:

- Built-in segmentation, memory mapping and attributes checking
- Efficient and cost effective operation
- Support for system-controlled relocation
- Support for stacks, with overflow checking
- Upward compatibility to 32-bit architecture
One of the most striking differences between the architectures of the Z8000 and 68000 is the provision for operating system support. This area received careful consideration in the Z8000 design. The designers of the 68000 addressed most of their careful attention to other issues. This paper shows the importance of operating system support features, even in relatively small applications, and contrasts the designs of the Z8000 and the 68000 in regard to operating system support.

OPERATING SYSTEMS

Every computer application contains an operating system—either explicitly or implicitly. For the purpose of this paper the following definition of an operating system is used:

The portion (hardware and software) of a computer application that is devoted to managing hardware and software resources.

Most definitions of "operating system" are similar to this one. The idea of resource management is central to everyone's idea of an operating system. The resources of a computer application can be divided (approximately) into the following categories:

- Processing elements (e.g., CPUs, floating point chips, "intelligent" disk controllers)
- Storage elements (e.g., ROM, RAM, disks, tape)
- External interfaces (e.g., I/O ports, modems)
- Programs (e.g., compilers, application programs)

A process (also called a task) is the ongoing execution of a program by one or more processing elements. For example, a compilation is a process. The goals of a computer application can be viewed as the completion of processes. From this point of view, the job of the operating system is to "direct traffic" for as many processes as it makes sense to run concurrently, allocating resources among these processes and resolving conflicts according to an externally selected policy. Directing traffic entails:

- Protection of the operating system and of each process from damage or invasion of privacy arising from the actions of any other process.
- Establishment, support, and enforcement of protocols and conventions for the interactions of system elements.
- Facilitation of interprocess communication and sharing.

Thus, the responsibilities of an operating system are:

- Allocation and protection of processing and storage elements, external interfaces, and programs.
- Definition, facilitation, and enforcement of protocols and conventions.
- Communication and sharing.
- Policy enforcement.

ARCHITECTURAL SUPPORT FOR OPERATING SYSTEM RESPONSIBILITIES

The operating system responsibilities listed above differ from system to system. For example, the work of a small application may be carried on by a single process, although the system process that handles external device interrupts will share the CPU with the application process. There are several kinds of architectural support that facilitate the operating system's task in a wide range of applications:

- Restriction of access to CPU facilities
- Restriction of memory use
- Memory mapping
- Sharing of programs and data
- Program relocation
- Stacks
- Context switching
- I/O system and interrupts
- Distributed control
- Support for conventions
Each of these features deals with one of the four responsibilities listed above—allocation and protection, protocols and conventions, communication and sharing, and policy enforcement.

Restriction of Access to CPU Facilities

The operating system must allocate the CPU to a process while still protecting itself and other processes. That is, the operating system must be able to turn the CPU over to a process and be assured that the process does not perform potentially destructive actions. A key to solving this problem is some kind of restriction of CPU use. Most CPU designs introduce a restricted "user" mode, in which certain instructions (called privileged instructions) cannot be executed and key CPU registers (called control registers) cannot be accessed.

The existence of a user mode and privileged instructions does not solve the entire protection problem. The other half of the solution involves restriction of access to memory and I/O. In addition, the introduction of user mode brings another problem, namely the question of how the CPU passes between system and user modes (especially, how it gets out of user mode). A solution frequently provided is one or more System Call instructions. These instructions allow programs running in user mode to call system mode programs without allowing the user mode program to retain control of the CPU once it has left user mode.

Restriction of Memory Use

Most CPU designs call for some sort of comprehensive memory management facility, which provides a unified approach to restriction of memory use, memory mapping, program relocation, sharing of programs and data, and stack use.

Restriction of access to memory usually depends upon the use of sets of attributes associated with portions of the memory address range of the CPU. These attributes are checked against certain access rights associated (implicitly or explicitly) with each process. Then, for example, if a program in user mode attempts to access a memory address whose attributes don't match the program's access rights, the CPU will trap to a system routine designed to deal with such invalid accesses.

The portions of the memory address range to which sets of attributes can be assigned depend upon the CPU addressing scheme and the memory management facility. Typically, in a machine using two-dimensional (segmented) addressing, attributes are associated with a segment. In a machine with linear addressing, attributes are usually associated with fixed-size blocks of addresses called pages. (See the 28000 vs. 68000 concept paper, "Segmented vs. Linear Addressing."

Memory Mapping

As noted above, the operating system allocates memory and programs and facilitates sharing and interprocess communication. These tasks are aided by memory mapping.

Memory mapping is the establishment of a function that assigns to each address (now called a logical address) in the memory address range an address in the actual physical memory available to the application. Naturally, a completely arbitrary function would be difficult to specify and to alter, so the usual approach is to divide the logical address space into blocks of contiguous addresses and to map each block to a block of contiguous physical addresses. All that is required to specify such a mapping is to provide the base physical address for each of these blocks, and, if not predetermined by the architecture, to provide the origin or sizes of the blocks of logical addresses.

In general, the blocks of logical addresses that can be mapped separately are the same as the blocks of memory that can be assigned attributes. The information about block size, base physical address, and attributes is usually grouped together into a segment or page descriptor.

Sharing of Programs and Data

Given memory mapping and access restriction, it is easy to see how the operating system can facilitate the sharing of programs and data among processes while still providing protection. For example, a block of physical memory containing a generally useful program can be "placed in the maps" of several processes. That is, each process can have a block of logical addresses (not necessarily the same addresses if the program is relocatable) that is mapped into the given block of physical memory. Similarly, a block of data can also be placed in the maps of several processes.

In the above examples of sharing, protection is provided by the access restriction mechanism discussed earlier. In the case of a program, for example, one of the attributes of the associated block of logical addresses can be that it is read only or even execute only. [This requires the existence of instructions and addressing modes that allow the generation of pure (i.e., not self-modifying) code.] Furthermore, the attributes can change, depending upon which process is running. For example, when the process responsible for a given block of data is running, the attributes of the associated block of logical addresses can be unrestricted, and when other processes are running, the attributes can include read only. The changing of attributes discussed here is part of the context switching accompanying the transfer of the CPU from one process to another. (In a multi-CPU system, the protection has to be provided by a difference in the access rights of the processes, not by a change in attributes.)
Program Relocation

There are three types of relocation: static relocation, dynamic logical address relocation, and dynamic physical address relocation. Static relocation is the kind provided by a relocating loader. Separate source files are assembled as if each were to begin at logical address zero, and a program combines these separate files into one large program designed to be run at fixed logical addresses. Dynamic logical address relocation is the process of changing the logical addresses at which a given program is to run. In most cases, this is possible only if the program has been designed to be independent of the logical addresses at which it runs. Dynamic physical address relocation is the process of changing the physical addresses at which a given program is to run, while leaving its logical addresses unchanged. By definition, dynamic physical address relocation makes sense only in connection with memory mapping.

Static relocation is possible regardless of the CPU architecture, whereas both kinds of dynamic relocation depend upon architectural support features. Both kinds of dynamic relocation help the operating system meet its responsibilities: Logical address relocation, as noted above, is important in program sharing, because it allows the logical address spaces of the processes sharing a given program to be managed independently. Physical address relocation is important in the allocation of memory, because it allows "garbage collection" to be performed easily and facilitates the implementation of virtual memory schemes.

Dynamic Logical Relocation. Dynamic logical relocation depends upon the ability to write programs that are independent of the logical addresses at which the program's instructions reside. Most CPUs provide some support for such programs. To understand this support, we must first understand what computer instructions do.

The CPU interprets instructions that are stored in memory. Each instruction must specify (explicitly or implicitly):

- The operation to be performed.
- The locations of the arguments.
- The location of the next instruction to be executed.

Computers have been designed (e.g., the IBM 650) in which all of these addresses are specified explicitly in each instruction. Obviously, no program on such a computer can be independent of the addresses at which the instructions reside. On most computers, however, position-independent means are available for specifying the locations of arguments and the sequence of instruction execution. These means all rely upon the use of registers and special addressing modes. The most fundamental of these registers is the Program Counter (PC), which is found in all modern computers; the associated addressing mode is called PC-Relative (or simply Relative) Addressing.

The PC contains the address of the next instruction to be executed. As each instruction is fetched from memory, the PC is changed to contain the address of the first memory location following the fetched instruction. Thus, in most cases, the sequence of instruction execution is defined by the sequence in which the instructions are stored in memory. This leaves only the case of transfer-of-control instructions (i.e., instructions that change the value of the PC) to be considered. Most modern computers have transfer instructions that add a signed offset (usually contained in the instruction) to the PC value. That is, these transfer instructions use relative addressing.

Obviously, transfer instructions that use relative addressing can be represented independently of the addresses at which the program containing the instructions resides. Other position-independent means of changing the PC are available on many computers:

- Indirect Register Addressing—the PC is set to a value specified in a register.
- Based addressing—the PC is set to the sum of an address value specified in a register and an offset specified in the instruction or in a register. (Many variations of based addressing exist.)
- Popping from a stack—the PC is set to a value previously saved on a stack.

Position-independent argument specification is achieved similarly. Based addressing, stacks, register addressing (the analog for argument specification of Indirect Register mode for transfers), and even Relative Addressing (for program constants) are commonly used ways of specifying arguments, which are available on many computers.

Dynamic Physical Relocation. Changing the physical addresses at which a program resides without changing the logical addresses is only possible with a memory-mapping scheme. Given memory-mapping, this kind of relocation requires no further architectural support.

Physical relocation is helpful to an operating system that must allocate a limited amount of memory among a varying set of processes or among processes with varying memory needs. It helps avoid the fragmentation of memory that can occur when there is dynamic allocation and release of varying amounts of memory. Physical relocation also is the basis of any virtual memory system.

A virtual memory system is an addressing scheme in which the logical address space is larger than the physical memory. Parts of the logical address space correspond to blocks of secondary storage, which are brought into physical memory only when a program attempts to access them. A virtual memory system requires extensive CPU support, since it
Stacks

Stacks are an important tool for meeting the operating system's responsibilities. A stack is a variably sized last in, first out memory. Associated with a stack are two operations, pushing (adding an item) and popping (removing an item).

Stacks are used by the operating system (explicitly or implicitly) to allocate memory in a flexible way that, in connection with based addressing, allows programs that need nonregister storage to remain position independent. Special cases of this are the storage of return addresses for subroutine calls and machine state for interrupt processing.

Stacks provide an important application of dynamic physical relocation, because the way they are used makes logical relocation of stacks almost impossible. In order to provide flexible allocation of stack space, the operating system must be able to expand a stack upon demand. This sometimes entails physical relocation of the stack to a larger area of physical memory, since with based addressing, a stack must consist of contiguous logical addresses and since most memory-mapping schemes require contiguous logical address blocks (below a minimum size) to map into contiguous physical addresses.

Other architectural features desirable for stack support include:

- The ability to designate one or more stacks for program use.
- Single- and multiple-argument push and pop instructions.
- The ability to address items at locations defined relative to the top of a stack.
- Automatic warning (traps) of impending stack overflow or underflow.

Most architectures call for the implementation of stacks as linear arrays in memory with an address register marking the top of the stack and providing (through based addressing) access to items at other locations in the stack. The stack register is a dedicated (special-purpose) register in some architectures. In other architectures, any address register can be used as a stack register, although the program usually cannot specify which stack register is to be used for saving returns from a subroutine or the machine state on interrupts.

The implementation of stacks as arrays in memory and the use of general-purpose address registers for stack registers make the provision of overflow and underflow protection difficult. Architectures that provide stack limit protection usually do so through the use of the attribute specification associated with memory protection. Several architectures provide stack access protection by means of a rudimentary separation of stack and data address spaces through externally interpreted CPU status outputs. A better approach is provided by a two-dimensional (segmented) addressing scheme, in which distinct objects and not just stacks can be assigned to independent parts of the address space. (See "Segmented vs. Linear Addressing," 28000 vs. 68000 Concept Paper.)

Context Switching

One of the difficulties of running several processes concurrently is the overhead associated with context switching. The context of a process is the portion of its state that occupies shared resources. For example, since most CPU architectures call for only a single Program Counter (PC), all processes must share this register, so the PC value of each process is part of its context. Most architectures also call for a single set of general-purpose registers, control registers, CPU status registers, and so forth. Thus, when the same CPU is allocated to more than one process, the process contexts must include the contents of any of these registers used by the processes.

Context switching is the saving of the context of one process and the recalling of the stored context of another process. Some architectural features for the support of context switching are desirable. These include automatic saving of CPU state on interrupts, single-instruction block register saving and restoring, and access to all necessary control registers.

All modern CPUs provide automatic saving of a portion of the CPU state on interrupts and access to all control registers that can form part of a process context. Block saving and restoring of registers is available with some CPUs. Either a starting register and the number of registers to be saved or a bit-encoded selection of registers to be saved provides some flexibility—not all registers need to be saved in every case. In most cases, the operating system saves registers on a stack.

I/O System and Interrupts

The operating system responsibilities pertaining to the I/O system and interrupts vary greatly with the type of application. The architecture of a general-purpose CPU must provide the flexibility necessary to accommodate the I/O requirements of a wide range of application types.

One of the operating system's most difficult tasks in this area is the control of access to I/O resources. Unlike memory, which can be divided into large, relatively homogeneous blocks, the elements of the I/O space require special-purpose management, protection, and access techniques. In addition, device timing requirements and externally set policies for conflict resolution make hardware support of I/O mechanisms mandatory.
Desirable architectural features for the support of the I/O system and interrupts include:

- A vectored interrupt scheme.
- Program-controlled specification of the CPU state to be established for each type of interrupt.
- A rapid, automatic context-switching mechanism for responding to interrupts.
- A means of defining conflict-resolution policies and "interruptibility" of interrupt processing.
- Block I/O instructions and DMA capabilities.
- Restricted access to I/O facilities.

A vectored interrupt scheme allows the CPU state to be switched immediately to an appropriate processing routine without the need for software to ascertain the interrupt type and call the appropriate routine. The port of connection or the contents of a vector supplied by the interrupting device are used to determine the new state.

A vectored interrupt scheme can be designed so that the new CPU state is specified in the hardware by the interrupting device, but in most architectures this is under program control. Most CPUs store this information in memory locations (often called interrupt vectors). In some CPUs a fixed block of addresses is devoted to storage of interrupt vectors, but a better approach is to allow any block of locations to serve and to have a CPU control register that points at the chosen block. One advantage of this approach is that the block of vectors can be assembled with the program and need not be set individually by initialization instructions. One disadvantage is that it discourages modular management of the vectors.

Every CPU with an interrupt facility has some kind of context-switching mechanism to support it, usually involving the use of a stack. In CPUs that support multiple stacks, the architecture designates one of these stacks for this use. Those parts of the machine state that the interrupt-processing routine cannot easily save by itself are pushed onto the designated stack. This saved information must include the PC value, and it can include other items as well. Usually, CPU condition indicators and operating mode bits are saved, while general-purpose register contents are not. Such CPUs have interrupt return instructions to pop the saved CPU state off the stack, thus returning the CPU to its pre-interrupt state.

Conflict resolution is controlled by a policy that is set by the system designer and enforced by the system. The usual approach is to provide a small number of priority levels to which device interrupts can be assigned, by virtue of either the means of connection to the CPU or the setting of a priority level in the "vector" for each device. Then, when the CPU is processing a device interrupt of a given priority level, only higher-level interrupts can occur.

Block I/O instructions and direct memory access (DMA) capabilities are important features that improve performance. Block I/O instructions require careful implementation. In general, they must use general-purpose registers to save their ongoing state, so that they can be interrupted. DMA capabilities require the development of bus control protocols and a means of protecting partially loaded or saved memory blocks from unwanted access by concurrently executing programs.

Restriction of access to I/O facilities can take many forms. In a CPU with a user operating mode and privileged instructions, the I/O instructions can be privileged. This is the easiest and most natural approach. In a CPU that does not have I/O instructions and a separate I/O address space (see 78000 vs. 68000 concept paper, "Memory Mapped vs. Explicit I/O"), a memory-protection approach must be taken.

Distributed Control

One of the recent advances in operating system design is the distribution of operating system functions among many separate processes. Such distributed systems present problems of interprocess synchronization.

When processes to which separate processing units may have been allocated share a common memory, the techniques of guarded commands and semaphores (developed by Dijkstra and others) are applicable. The basic architectural support for these techniques is the atomic Test and Set instruction, a CPU instruction that tests a memory location for the value "available" and simultaneously sets the value to "not available." The word "atomic" means that there can be no other access to the given memory location between the "test" and "set" portions of the instruction, so no two concurrently running processes can find the location set to "available" simultaneously. Implementation of a Test and Set instruction requires a bus-locking mechanism.

When processes do not share a common memory, a similar nonmemory exclusion mechanism must be provided. A separate bus can carry the signals needed to implement such a mechanism, and CPU instructions can be provided to manage the CPU's connection to that bus.

Support for Conventions

One of the issues that must be considered in the design of a CPU is whether its architecture should support all conventions equally, favoring none, or whether it should encourage, through special features, specific conventions. For example, should a CPU be designed with general support for high-level languages, or should it be designed to optimize Pascal, say, at the expense of making FORTRAN programming less efficient? Should it provide special features that make a subroutine
argument passing convention using the stack especially efficient at the cost of decreased efficiency for other argument passing conventions.

In practice, there are many cases in which the choice to support one method of accomplishing a task makes the designer's job easier, but discourages the use of other, equally valid approaches. If the other approaches are no better than the one supported, then support of one specific approach is a net advantage. But if the unsupported approach is preferable to the supported one in some applications, and if the special support feature makes the unsupported feature less efficient than it would otherwise have been, then there is a net disadvantage for those applications.

Another aspect of the system's support for conventions is the definition of the CPU's operating environment provided by a coherently designed family of components and a compatible interconnect bus. In most CPU architectures, this definition of the CPU's operating environment is not given much attention. Key points that should be considered are:

- The need for a staged, modular development--over many years--of a CPU and its component family.
- The importance of changing the distribution of function between the CPU and associated components with minimal impact on existing programs.
- The need for future enlargement of capabilities without substantial redesign of existing components or systems.

**THE Z8000 APPROACH**

The Z8000 designers were aware of all of the operating system support features mentioned in the preceding section, and an attempt was made to provide for these support features in the Z8000 architecture. Naturally, other design criteria were present and tradeoffs were made, but on the whole, a better and more unified approach to operating system support was taken with the Z8000 than with other CPUs in its class.

**Restriction of CPU Use in the Z8000**

The Z8000 has a system/normal bit in its Flag/Control Word register (FCW). When the bit is in the normal state, privileged instructions cannot be executed. Operating system tasks are expected to execute in the system mode. The privileged Z8000 instructions are:

- I/O instructions, including the interrupt return and nonmemory synchronization instructions.
- Control register manipulation instructions.
- The Halt instruction.

In addition to privileged instructions, another protection feature is associated with the system/normal bit. There are two copies of the implied stack register (the stack register used for interrupt and subroutine returns); one is used when the CPU is in system mode, the other when it is in normal mode. Programs executing in normal mode have no access to the system mode stack register.

Passing between system and normal modes requires a change to the FCW. This can only be accomplished through a privileged instruction (LDCTL, IRET, LDPS) or automatically in response to an interrupt or trap. A system call trap (a one-word instruction with eight programmable bits) allows a normal mode program to call one of 256 system mode programs.

**Memory Management in the Z8000**

The Z8000 design provides for a comprehensive memory management facility, which offers a unified approach to restriction of memory use, memory mapping, sharing of programs and data, program relocation, and stack use. This memory management facility is integrated with a segmented (two-dimensional) addressing scheme in the CPU. (For a discussion of this entire area, see the Z8000 vs. 68000 concept paper, "Segmented vs. Linear Addressing.") One of the many advantages of segmentation is that it is an ideal organization for a system (e.g., UNIX*) in which there are many small tasks.

Another feature of the Z8000 memory management facility is that it is designed to facilitate the implementation of virtual memory systems. Virtual memory is an important technique for handling the enormous address space of a CPU like the Z8000 with a reasonable amount of physical memory. Details of the implementation of Z8000 virtual memory systems are available in the articles on the Z8003/Z8004 by Calahan, Patel, and Stevenson and on the PMMU by Hu, Lai, and Stevenson (both to appear in "Electronics" in late summer 1981).

**Context Switching in the Z8000**

Z8000 interrupt and trap-handling provides an automatic, rapid context switch from the executing program to the interrupt-processing routine. The FCW and PC values and a "reason" are saved on the system mode stack, and new FCW and PC values are set from the program status area (PSA) entry corresponding to the interrupt type.

The Z8000 block register saving and restoring instructions facilitate context switching. These instructions can be used to simulate the pushing or popping of a block of registers to or from any stack.

* UNIX is a registered trademark of Bell Laboratories.
In some cases, the values of control registers are essential to the context of a process. (The normal stack register and the FLAGS register are obvious examples.) A load control register instruction allows the transfer of any of these registers to or from a general-purpose register, so they can be saved and restored like other registers.

The Z8000 Component Family and the Z-BUS

A fundamental concept in the Z8000 architecture is that of a family of components designed to work together. Because a CPU and its associated component family must be developed and introduced over a span of several years, the outlines of the family and the framework to support it must be established at the outset. In the case of the Z8000, the CPU chip was designed with many hooks in place, including segmented memory addressing, nonmemory synchronization instructions, block I/O instructions, a multiplexed address/data bus, an encoded 4-bit CPU status output, and extended processor instructions. The extended processor instructions are an especially good example of this planning. Processor instructions and a bus protocol allow for the development of "slave" processors (like a floating-point chip) that execute instructions taken from the CPU's instruction stream and have access to the CPU's addressing capabilities.

The Z81TM Z-BUS Component Interconnect provides the signal lines and protocols required to tie members of the Z8000 family together and provides the necessary interface specification for family members still to be developed.

An even wider environment for the CPU is defined by the Z81TM Z-BUS Backplane Interconnect, which is compatible with the ZCI and provides for expansion of the Z8000 to a full 32-bit architecture.

The Z8000 I/O System and Interrupts

The Z8000 uses a block of memory called the program status area (PSA) to store interrupt vectors (i.e., the new CPU status) for each type of interrupt and trap. In addition to separate lines for nonvectored and vectored interrupts and a nonmaskable interrupt for situations that can't wait, there is a table of PC values to be indexed by an 8-bit vector placed on the address/data bus by the interrupting device. The block of memory used for the PSA is not fixed, as with some CPUs it can be anywhere in memory, and a pointer to it (the PSAP register) can be set using the privileged LDCTL instruction.

Conflict resolution is done simply. The three kinds of interrupt (nonmaskable, nonvectored, and vectored) are assigned three levels of priority by the CPU. In addition, the vectored and nonvectored interrupt lines can be masked (using the privileged Disable/Enable Interrupt instruction), so that interrupts are not processed until the unmasking of the associated line. When interrupts arrive on more than one line simultaneously, the priority determines which is processed first. The processing routine for any interrupt type can be interrupted by the routine for any other if the corresponding line has not been masked. Whether other lines are to be masked or not can be determined automatically by specifying the appropriate mask bit in the FCW portion of the PSA entry. Otherwise, the determination can be made by the program, which can bracket sensitive code between Disable Interrupt (DI) and Enable Interrupt (EI) instructions.

A daisy chain is used to determine the order of processing of interrupts from devices attached to the CPU on the same interrupt line. In this way, devices closer to the CPU can interrupt the processing of interrupts from devices farther away from the CPU, unless the given line is masked during all or part of the processing.

A key aspect of the Z8000 I/O system is the protection provided by privileged instructions. This protection allows an operating system to manage the I/O interfaces without interference from normal mode programs.

Distributed Control

The Z8000 architecture provides ways to synchronize processes that share memory and those that do not. The Test and Set instruction provides the basis for synchronization of processes that share memory. For nonmemory synchronization, the Z-BUS has a set of lines and a protocol for resolving simultaneous requests for shared resources, and the CPU provides instructions to support the bus connection and protocol.

Support for Conventions

The Z8000 design supports many conventions. Principal among these are

- Use of a segmented address scheme.
- Use of message passing for interprocess communication.
- Component and backplane bus protocols.
- Interrupt protocols for all components.

The only convention listed above that has not yet been discussed is message passing. A message is a set of characters sent by one process and received, asynchronously, by another. The processes do not need to know whether they have been allocated the same or different processing elements.

The Z8000 family architecture provides message passing support both in the CPU and in other components.
- Block I/O instructions in the Z8000 CPU support message passing.
- The Z-FI0 (FIFO I/O unit) chip provides the asynchronous interprocessor connection necessary to a message-passing philosophy.
- The Z-UPC (Universal Peripheral Controller) chip accepts commands from and delivers messages to the master CPU in designated message registers in the Z-UPC.
- The DMA (Direct Memory Access) chip has been designed with a "flyby" mode that allows external devices (e.g., a Z-FI0 chip) high-speed direct access to memory without storage of the data by the DMA chip.

**THE 68000 APPROACH**

The 68000 provides many of the architectural support features mentioned earlier, but there are several that did not receive the same careful attention given to the Z8000 design.

**Restriction of CPU Use in the 68000**

The 68000 has system and user modes and privileged instructions, just like the Z8000. The two main differences are:

- In the Z8000 I/O instructions are privileged, while in the 68000 ordinary memory reference instructions double as I/O instructions and thus cannot be privileged.
- The Z8000 has one System Call (SC) instruction with 8 programmable bits, while the 68000 has 16 separate System Call instructions, none of which has any programmable bits.

The 68000 operating system designer is forced to use an external memory management system to implement the protection of I/O operations. The Z8000 operating system designer can work with privileged instructions—a tool that is consistent with the tools used for protection of other key Z8000 functions. (For a detailed discussion see the Z8000 vs. 68000 concept paper "Segmented vs. Linear Addressing".)

The second point boils down to the number of distinct instructions available for system calls and the number of separate traps over which these instructions are distributed. The 68000 architecture provides for a total of 16 system calls and ties them all to separate traps. The Z8000 architecture provides for 256 system calls and ties them all to one trap, so that dispatch software is required to route the calls to the proper routines, but obviously, the Z8000 design can accommodate the addition of a hardware dispatch mechanism in the future with no change to user programs. Furthermore, the 68000 approach forces duplication of context-saving operations (e.g., register saving).

The key difference is that the Z8000 has 256 System Call instructions, while the 68000 has only 16. The Z8000's 256 calls will accommodate the needs of most applications. The 68000's 16 calls will be too few for all but the simplest cases, so most 68000 applications will be unable to use the system and user modes in a natural way.

**Memory Management in the 68000**

The key point to understand about the 68000 memory management mechanism is that it is completely external to the CPU. Several facts follow from this:

- The 68000 cannot use segmentation without extensive software overhead.
- Because of the overhead that a fully general mapping facility would entail, the 68000 must use a "simple but powerful" scheme that limits mapping to the bitwise replacement of fields in the address.

Naturally, many benefits can be derived from the use of such a scheme, but it can never be more than a separate, after-the-fact transformation and checking mechanism, so that the 68000 is not relieved of the problems of linear addressing. (See the Z8000 vs. 68000 Concept Paper "Segmented vs. Linear Addressing".)

**Context Switching in the 68000**

There is very little difference between the Z8000 and the 68000 in their approaches to context switching.

**The 68000 Component Family and Bus**

The idea of a family of components designed to work together, which is so fundamental to the Z8000 philosophy, is not clearly evident in the 68000 design. While the Z-BUS forms the framework for the entire, expanding Z8000 Family, the 68000 seems to have been designed with an eye toward compatibility with the older 6800 family peripherals and with existing bus structures.

While the Z8000 is designed to include features to facilitate its integration into a family of components, the 68000 seems to have been designed before there was a clear conception of what its environment would be. For example, the Z8000 has provision for memory management integrated into the CPU; the 68000 memory management mechanism is entirely external to the CPU. The Z8000 has a nonmemory synchronization facility (no bus or processor provision for nonmemory synchronization exists in the 68000). The Z8000 was designed with a multiplexed address/data bus to accommodate the advanced programmable peripherals designed with it; the 68000 was designed with a nonmultiplexed address/bus (See the Z8000 vs. 68000 Concept Paper "Multiplexed vs. Non-Multiplexed Address/Data Bus"). The Z8000 was designed with block I/O instructions to facilitate the message passing protocols to be used with the Universal Peripheral Controller and, via the FIFO interface, with other
processors and devices; the 68000 does not seem to support any particular interprocess communication protocols and techniques.

These examples only serve to emphasize the key point: the Z8000 design is based on a family concept, the 68000 design is not.

The 68000 I/O System and Interrupts

The Z8000 and 68000 interrupt systems are similar, with roughly equivalent capabilities. Notable differences are:

- Location of the interrupt vectors. In the Z8000, the PSA can be anywhere in program memory. In the 68000, the interrupt vectors must be specific locations in data memory.
- Resolution of priority. The 68000 requires each device to supply a 3-bit interrupt request code that is used in determining the device's priority. In the Z8000 family, devices can be used at any priority level without modification. Each device is attached to one of three interrupt request lines, and each line has a different priority. A daisy-chain protocol defines priorities among devices attached to the same line.

A key difference between the Z8000 and 68000 I/O systems is the use of explicit I/O instructions in the Z8000 and the use of memory mapped I/O in the 68000. (See the Z8000 vs. 68000 Concept Paper "Memory Mapped vs. Explicit I/O".) Among other problems, this forces the 68000 to use the coarse-grained protection of memory management to do what the Z8000 accomplishes with privileged I/O instructions.

Distributed Control

The 68000, like the Z8000, has a Test and Set instruction for synchronizing processes that share memory. The 68000 lacks a mechanism for nonmemory synchronization such as is provided by bus protocols and CPU instructions on the Z8000.

Support for Conventions

The only area in which the 68000 provides support for an operating system's definition of conventions is subroutine argument passing. By means of its Link and Unlink instructions and its stack-oriented addressing modes, the 68000 design encourages a stack-based argument-passing scheme. While the Z8000 also allows an efficient stack-based argument-passing convention, it provides equally good support for a register-based convention.

In the areas of memory addressing, component and backplane bus protocols and interprocess communication, the 68000 provides little support for the framework of conventions that an operating system must provide. Especially striking is the contrast between the Z8000's system-wide support of message passing for interprocess communication and the 68000's failure to provide any special support for interprocess communication.

SUMMARY

Operating systems are responsible for the allocation and protection of processing and storage elements, external interfaces and programs; for the definition, facilitation and enforcement of protocols and conventions; for communication and sharing; and for policy enforcement.

Several kinds of architectural support help operating system designers meet these responsibilities: restriction of access to CPU facilities, restriction of memory use, memory mapping, sharing of programs and data, program relocation, stacks, context switching, an I/O system and interrupts, distributed controls and support for conventions. Both the Z8000 and the 68000 provide these kinds of support, but the Z8000 approach is more integrated and far-reaching.

The most notable differences between the Z8000 and the 68000 are:

- The support for virtual memory in the Z8000.
- The lack of privileged I/O instructions and the scarcity of System Calls on the 68000.
- The lack of a family concept in the 68000 design, and the resulting lack of cohesion among the 68000 and its peripheral components.
- The Z8000's greater flexibility in the specification of interrupt vectors and the determination of device priorities.
- The lack of a provision for nonmemory process synchronization in the 68000 design.
- The absence of support for message passing (or any other interprocess communication scheme) in the 68000 design.

The Z8000 is seen to provide superior support for operating system functions.
Exploring 16-bit µCs

A tale of four µPs:
Benchmarks quantify performance

Aided by portions of the Carnegie-Mellon test package and with the cooperation of DEC, Intel, Motorola and Zilog, EDN compares the performance of four popular processors.

Robert D Grappel, Consultant, and Jack E Hemenway, Consulting Editor

In this article, EDN proudly publishes the results of the first comprehensive benchmark study of four major 16-bit processors: the Digital Equipment Corp LSI-11/23, Intel 8086, Motorola 68000 and Zilog Z8000. You'll find these results highly interesting, and they should help you choose the best device for your application.

Don't assume, however, that limiting the study to four devices implies that they are the four "best" machines; we hope that future articles will add new processors to the comparisons. Nevertheless, any benchmark study must start somewhere, and these machines seem representative of those used in today's systems.

Why the need for a benchmark study at all? One sure way to start an argument among computer users is to compare each one's favorite machine with the others. Each machine has strong points and drawbacks, advantages and liabilities, but programmers can get used to one machine and see all the rest as inferior. Manufacturers sometimes don't help: Advertising and press releases often imply that each new machine is the ultimate in computer technology. Therefore, only a careful, complete and unbiased comparison brings order out of the chaos.

The benchmarking process isn't a new one; Special Features Editor Robert Cushman has explored much the same ground for older processors that this article does for newer ones (EDN, April 20, 1975, pg 41). The modern devices are more powerful, but the task of choosing the right one for a given application is no simpler.

Who chose the benchmarks?

Benchmarking anything as complex as a 16-bit processor is a very difficult task to perform fairly. The choice of benchmark programs can strongly affect the comparisons' outcome, so the benchmark must choose the test cases with care. The programs used in this article were compiled in 1976 by a group at Carnegie-Mellon University for use in benchmarking minicomputers and mainframes. The group presented the results of those benchmark tests at the 1977 National Computer Conference in the paper "Evaluation of Benchmark A—I/O interrupt kernel

The test case chosen for this benchmark consists of one interrupt at each of the four levels. The times shown are the sum of the four interrupts, computed by counting the number of the instruction cycles by hand and including the time required for the processor to recognize and process an external interrupt.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Clock Speed (MHz)</th>
<th>Code Bytes</th>
<th>Execution Time (µsec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSI-11/23</td>
<td>3.33</td>
<td>20</td>
<td>114</td>
</tr>
<tr>
<td>8086</td>
<td>10.00</td>
<td>55</td>
<td>126 (note)</td>
</tr>
<tr>
<td>68000</td>
<td>10.00</td>
<td>24</td>
<td>33</td>
</tr>
<tr>
<td>Z8000</td>
<td>6.00</td>
<td>18</td>
<td>42</td>
</tr>
</tbody>
</table>

Note that the 8086 implementation of this benchmark saves the complete machine context on the stack; it's the only implementation that does so.

Fig 1—This benchmark tests a processor's basic interrupt capability. Some machines require external hardware, while others put limitations on the number of interrupt vectors supported.
When faster chips are developed, execution times will decrease

Computer Architectures via Test Programs," by S H Fuller, P Shaman, D Lamb and W E Burr.

The set of programs includes many common algorithms that appear frequently in real-world applications. They test the ability of a computer to handle data in chunks ranging from individual bits to 32-bit integers to floating-point numbers. The tests include interrupt handlers and character-string searches, bit manipulations and sorting. Taken as a set, they encompass a fair test of a computer's real power.

For our benchmark tests, we have chosen a subset of this Carnegie-Mellon set; we have not included the benchmarks dealing with floating-point math or virtual-memory handling. We excluded floating-point math because most of the 16-bit processors don't support floating-point operations; we would thus have ended up benchmarking their floating-point software or external math processors. Similarly, the processors don't provide virtual-memory support. We also excluded two benchmarks that require extensive number-crunching capability (Fourier transforms and Runge-Kutta integration) because their results would depend so heavily on the floating-point support used. The remaining seven benchmarks (labeled A, B, E, F, H, I and K in the Carnegie-Mellon literature) provide a sufficient test of each processor without handicapping any of the contestants.

Who coded the benchmarks?

Clearly, once you've chosen a benchmark set, coding it is the next critical task. We wanted to show each machine in the best possible light. Therefore, we asked a representative of each manufacturer to code the set for that manufacturer's machine, assuming that the manufacturer would best understand its machine's features.

To referee this process, EDN then reviewed each program. Additionally, we circulated copies of each program for comments to the programmers of each of the other computers. This process ensured careful proofreading and close adherence to the benchmark-test rules.

We did allow minor variations in the benchmark implementations where we felt they were justified. Two of the processors (the 8086 and Z8000) have special character-string instructions—we didn't want to penalize these devices by forcing their manufacturers to code unnecessary loops. Additionally, the 8086 has no general bit-manipulation instructions—a feature that sometimes produced differences among the bit-level benchmark implementations. The processors also differ widely in their extended-addressing capabilities, so we allowed some latitude in addressing mechanisms.

The manufacturers coded each benchmark in assembly language, for several reasons. First, of course, there is no mutually acceptable high-level language available on all the machines. But in any case, EDN wanted to benchmark the processors and not the capabilities of compiler writers. The documentation provided to each programmer was the Carnegie-Mellon specification, in the form of flowcharts or a PASCAL-like pseudocode. The representatives required several iterations to get all the programs to work and then to optimize them; the final code appears at the end of this

Benchmark B—f0 function with FIFO processing

The test data for this benchmark consists of the following set of interrupts:
- Level 1 once
- Level 2 once
- Level 3 once
- Level 4 once
- Level 2 three times (forces queueing of interrupts)
- Level 3 five times.

The times shown are the sum of all of these interrupts. We hand-computed these times by counting instruction cycles, assuming worst-case arrival times.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Clock Speed (MHz)</th>
<th>Code Bytes</th>
<th>Execution Time (usec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSI-11/23</td>
<td>3.33</td>
<td>86</td>
<td>1196</td>
</tr>
<tr>
<td>8086</td>
<td>10.00</td>
<td>85</td>
<td>348</td>
</tr>
<tr>
<td>68000</td>
<td>10.00</td>
<td>118</td>
<td>390</td>
</tr>
<tr>
<td>Z8000</td>
<td>6.00</td>
<td>106</td>
<td>436</td>
</tr>
</tbody>
</table>

Fig 2—Extending Benchmark A, this test includes a FIFO buffer that queues interrupts as they arrive. The 68000 and 28000 FIFO implementations trade a few words of memory for a fast and simple queue structure.
article.

We provide the complete source-code listing of each benchmark on each machine for two reasons. First, readers can then duplicate the benchmarks on their own machines. Many “benchmark” numbers have appeared in print as new processors are introduced, but without a display of the coding, these numbers have no real basis. Second, a good program’s “flavor” and “style” reveal much about the way you should program a particular processor. And these are carefully written and carefully checked programs produced by experienced programmers.

An I/O interrupt kernel tests interrupts

With this background information in mind, you can examine each benchmark test. Benchmark A tests a computer’s interrupt-handling capability; the flowchart in Fig 1 illustrates the task to be performed. An I/O interrupt with priority 0, 1, 2 or 3 occurs from one of four devices. The actual interrupt handler merely counts the device-interrupt occurrences.

The interrupt handler must be able to pre-empt processing lower priority interrupts and must provide for resumption of the processing of lower level interrupts from the point of pre-emption. As much processing as possible should occur with interrupts enabled.

This benchmark must take into account the existence of interrupt-prioritizing hardware, whether on the processor itself or as added peripheral components. Some of the steps in the flowchart are automatic in some machines.

Benchmark B (Fig 2) also assumes four interrupting devices, except that here the interrupts are handled on a first-in, first-out basis rather than by priority. A queue with space for at least 10 pending interrupts must be provided. Processing of queued interrupts occurs with I/O interrupts enabled; thus, interrupts are accepted and queued appropriately while previous interrupts are processed. Before returning to the originally interrupted application program, the code must check whether any interrupts remain queued; if so, it must process them in FIFO order.

Benchmark E appears in PASCAL-like form in Fig 3. This familiar task searches a text string for the existence of a text substring. If the string search is successful, the routine returns the substring’s position in the data string. Otherwise, it returns a “not-found” indicator. A satisfactory program for this task must be re-entrant and position independent.

Benchmark F (Fig 4) checks a processor’s primitive bit-manipulation capabilities. It assumes a tightly packed bit string starting on a word boundary. A function code (F) chooses the appropriate operation from among Test (F=1), Set (F=2) and Reset (F=3). Bits are numbered relative to the bit string’s starting address. This program, too, must be re-entrant and position independent.

Linked-list insertion—a common problem

Benchmark H (Fig 5) deals with inserting new entries into a doubly linked list. The Key field in each entry is a 32-bit integer value. This benchmark exercises a processor’s addressing capabilities and also checks its ability to compare 32-bit quantities.

Each entry in the list has a Key and a forward and backward pointer: The first list entry’s backward pointer is zero; the last entry’s forward pointer is zero. New entries must be inserted in ascending order of their Keys. The benchmark assumes the existence of a list-control block (LISTCB) that holds a pointer to the first entry in the list (HEAD), a pointer to the last entry in the list (TAIL) and a count of the number of entries in the list (NUMENTRIES). Like programs for Benchmarks E and F, this one must be re-entrant and position independent.

Benchmark I (Fig 6) is the well-known Quicksort algorithm, which tests a processor’s ability to manipulate stacks and also gives the device’s addressing modes a workout. This is by far the most complicated benchmark in the set, and manufacturers’ coding of it exhibits the widest variation.

The benchmark specifies N 16-byte records. The data array REC actually holds N+2 records, with REC0 holding the lowest key value and RECN+1 the highest. A program must sort the records, based upon a key formed from the characters in each record’s third through ninth bytes. Hence, this benchmark also tests character manipulations. Parameter M specifies the changeover point between Quicksort and a simple insertion sort. This program, too, must be re-entrant.
Each manufacturer coded the programs for its processor

and position independent.

Finally, Benchmark K (Fig 7) transposes a matrix of bits. This test further checks out a processor's bit-manipulation facilities, as well as exercising loop constructs. It assumes a tightly packed bit matrix starting on a word boundary, and its program must be re-entrant and position independent.

On your marks, get set, ...

Before examining the programs themselves, pause and recall the µP state of the art a scant few years ago. A programmer would have been hard pressed to write many of these benchmarks on the 8-bit µPs available then; for one thing, the requirements of position independence and re-entrancy would have caused nightmares. In this light, the power of all the processors described in this article is very impressive.

Turning now to the benchmark programs, we note that all of them appear to be properly coded and bug free. However, some differences among them deserve comment.

There's a major difference in "philosophy" between machine designers and programmers who favor registers and those who favor memory—a fact that's especially clear in the case of passing parameters to subroutines. Note in this regard that the various processor manufacturers are not especially consistent in this area. The LSI-11/23 programs use the stack for parameters; the 68000 and 8086 sometimes use the stack and sometimes use registers; the Z8000 uses registers exclusively. Because our benchmark specifications merely say "re-entrant," either method is satisfactory, provided that the programs furnish register-save and -restore instructions. Using the stack is "cleaner," but registers are usually faster.

A problem of benchmark interpretation is apparent for the specification of Benchmark A: What does "Save Context" mean? One reading would suggest making a copy of the machine state (registers, program counter, condition codes, etc) while another might argue that all the specified calls for is saving the contents of any registers actually used (assuming that the actual interrupt-processing routines save and restore registers). Clearly, you get more compact code and faster execution times by choosing the "save only what is used" approach; only Intel performed an explicit and complete context-save operation.

Also observe that several of the interrupt benchmarks use special hardware. The 8086 version, for example, assumes that an 8259 PIC chip is available to field and prioritize interrupts. And the LSI-11/23 has fully vectorized interrupt hardware. (Thus, the interrupt benchmarks would run slightly slower on an LSI-11/2, because the processor is without such hardware.) The Z8000 assumes device device chaining, while the 68000 uses its built-in vectoring.

Finally, note the ONTRACE and OFFTRACE instructions in some of the LSI-11/23 code. DEC has used these commands to trigger a logic analyzer at the start and end of each routine being timed. Other benchmark programs also contain instructions intended for timing purposes. We didn't include these instructions in our code-byte totals, though.

### Benchmark F—Bit set, reset, test

The test data for this benchmark is an array of 125 bits consisting of alternating ZEROs and ONEs. The array begins on a word boundary. The times shown are the sum of the following nine tests:

<table>
<thead>
<tr>
<th>Test</th>
<th>Function</th>
<th>Bit Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TEST</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>TEST</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>TEST</td>
<td>123</td>
</tr>
<tr>
<td>4</td>
<td>SET</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>SET</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>SET</td>
<td>123</td>
</tr>
<tr>
<td>7</td>
<td>RESET</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>RESET</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>RESET</td>
<td>123</td>
</tr>
</tbody>
</table>

Note that the processors should perform these tests in this order without resetting the bit string. Motorola and Intel hand-computed their times; the others come from actual computer runs with real-time clocks or logic analyzers.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Clock Speed (MHz)</th>
<th>Code Bytes</th>
<th>Execution Time (μsec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSI-11/23</td>
<td>3.33</td>
<td>70</td>
<td>799</td>
</tr>
<tr>
<td>8086</td>
<td>10.00</td>
<td>48</td>
<td>122</td>
</tr>
<tr>
<td>68000</td>
<td>10.04</td>
<td>36</td>
<td>70</td>
</tr>
<tr>
<td>Z8000</td>
<td>6.00</td>
<td>44</td>
<td>123</td>
</tr>
</tbody>
</table>

```
procedure BITTEST(F,N,Al,RC,WORK)
  integer ABIT,D

  ABIT := Al+N/(word length)
  D := N mod (word length)

  if Dth bit at address ABIT=1
     then RC := 1
     else RC := 0
     end-if

  if F = 2
     then Dth bit at address ABIT := 1
        else if F = 3
           then Dth bit at address ABIT := 0
           end-if
     end-if
```

Fig 4—Benchmark F exercises each processor's bit-manipulation capabilities. The 8086 and Z8000 use a somewhat different algorithm than the other two devices; they always test the specified bit, regardless of the function code. If they then find the bit to be ZERO and the function to be Reset, they merely return. On the other hand, if they find the bit to be ONE and the function Set, they also return. Finally if the function is Test, they can return regardless of the bit value. This version of the algorithm saves some execution time at the expense of code that's not as clear. The 8086 doesn't have the bit-manipulation instructions of the other processors, so it must use shifts and other instructions to perform the bit-manipulation operations.
**Benchmark H—Linked-list insertion**

This data set starts with an empty list, into which five records are inserted with keys (32-bit hexadecimal numbers), as shown. The timings are for the sum of all five insertions:

1. 12345
2. 12300
3. 13344
4. 12345
5. 34126

<table>
<thead>
<tr>
<th>Processor</th>
<th>Clock Speed (MHz)</th>
<th>Code Bytes</th>
<th>Execution Time (μsec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSI-11/23</td>
<td>3.33</td>
<td>138</td>
<td>592</td>
</tr>
<tr>
<td>8086</td>
<td>10.00</td>
<td>94</td>
<td>—</td>
</tr>
<tr>
<td>68000</td>
<td>10.00</td>
<td>106</td>
<td>153</td>
</tr>
<tr>
<td>Z8000</td>
<td>6.00</td>
<td>96</td>
<td>237</td>
</tr>
</tbody>
</table>

Motorola hand-computed the 68000 timings; the remainder come from real-time clocks or logic analyzers.

if PRESENT.PREV = 0 and NEW.KEY < PRESENT.KEY
then * new list head *

LISTCB.HEAD := NEW
PRESENT.PREV := NEW
NEW.NEXT := PRESENT

else if NEW.KEY < PRESENT.KEY
then * new list tail *

PRESENT.NEXT := LISTCB.TAIL := NEW
NEW.NEXT := PRESENT
PRESEN.T.PREV := NEW

else *insert in middle*

NEW.NEXT := PRESENT
NEW.PREV := PRESENT.PREV
PRESEN.T.PREV := NEW

* back up and link predecessor *

PRESENT := NEW.PREV
PRESENT.NEXT := NEW

* end-if

Fig 5—Linked-list insertion using a 32-bit key value tests many aspects of a 16-bit processor's architecture. This benchmark exercises the addressing modes of each device.

**Memory limitations surface**

A major feature of the new 16-bit processors is their ability to address large memories. Unfortunately, many of the benchmark programs were coded in a way that limits them to a 64k-byte range; only the Motorola 68000 programs are truly usable over the machine's full addressing range.

The LSI-11/23 and Z8000 benchmarks assume a 64k data space, because they use only 16-bit addressing. For example, in Benchmark E, the character-string search, neither of these machines can (with the coding shown) deal with the case where the search substring and the data string are not in the same 64k space. In that case, you would require additional coding to handle the segment information, necessary to extend the programs to the processor's full addressing range.

In the same vein, the 8086 benchmark programs frequently assume that the calling program and the subroutine share data and stack segments—an assumption that also limits the subroutine's addressing range. For example, the character-string-search Benchmark (E) assumes that the string to be searched is in the extra segment (ES). This must be the case to make the compare-string (CMPS) instruction work properly; if the string were not already in the extra segment, you would need code to change the segment addressing.

The 8086 coding of the Quicksort (Benchmark I) uses a clever trick involving the 8086 segment registers to gain efficient indexing of the data records. Unfortunately, this trick only works because the records are exactly 16 bytes long. Because the 8086 addressing system internally multiplies each segment by 16, putting a record number in the segment register automatically points to the appropriate address. Executing Quicksort for records of any other length, though, would require rewriting the Intel program. Modification of this routine for general record lengths would increase code size by an estimated 25% worst case (this also allows records extending over segment boundaries) while affecting performance by no more than an estimated 5%. The performance degradation occurs only for segment-boundary checks, record-length incrementing through the array (rather than segment-register incrementing) and segment-boundary transitions. The code expansion arises from segment-boundary-transition logic that's infrequently— if ever— invoked.

Note that the Zilog benchmarks shown are coded for the Z8002 (unsegmented) version of the Z8000. On the segmented (Z8001) version, these programs would be virtually identical: Except for the I/O-interrupt-kernel benchmarks, all of the programs use exactly the same number of bytes for both devices. (The I/O-interrupt-kernel routines use direct addressing for some variables.) Execution times for the Z8001 benchmarks would tend to be longer than the Z8002 times, though, because of such factors as:

- 32-bit Load instructions for address moving
Tests attempt to measure μPs, not programmers’ skills

- More registers to save and restore
- The longer execution time of the RET instruction
- The longer time required for direct addressing.

What did we measure?

Two statistics are important in computer benchmarks: program size and speed. A program’s size is easy to measure—just add up the bytes. Our ground rule in this regard was “If you placed the program in ROM, how much ROM would be used?” We didn’t count stack space. (There are no local variables, because the benchmarks are re-entrant.)

Speed values, on the other hand, are very difficult to get a handle on: It seems that the chip makers produce faster μPs weekly. The memory you use can also affect the execution speed, thanks to such factors as dynamic-memory refresh. Therefore, because it wasn’t possible to obtain a consistent timing mechanism for all of the benchmarks, the timing information provided is merely what the programmers themselves measured.

(or in Motorola’s case, calculated). We do include data on the processors’ clock rates, as well as how the timings were obtained. And we also performed spot checks on the timing figures provided, using our experience in working with these processors to ensure that the times were reasonable.

Execution times for the 8086-, Z8000- and 68000-based single-board computers assume on-board memory-access operations. By contrast, results for the LSI-11/23 are based on the use of standard off-board dynamic-RAM systems and an asynchronous bus for instruction and data transfers—a configuration dictating the use of processor Wait states, which slowed speeds somewhat. DEC points out, however, that the LSI-11/23’s performance figures reflect the actual operation of current board-level product offerings and that the data doesn’t necessarily reflect a limitation of the board’s processor chip set.

Finally, note that we list the clock speeds of the fastest boards currently available; ie, we have 10-MHz units from Intel and Motorola running in our lab. However, we expect that the manufacturers will build even faster machines in the future. For example, Zilog plans to introduce a 10-MHz version of the Z8000 within the next 3 months. Because faster processors obviously

---

**Benchmark I—Quicksort**

The test data for this benchmark consists of 102 (N=100) records, each 16 bytes long. Parameter M is set to nine. The records are initialized as follows:

<table>
<thead>
<tr>
<th>Record</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>1</td>
<td>FF 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>2</td>
<td>FE 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>3</td>
<td>FD 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>100</td>
<td>9C 00 00 00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>101</td>
<td>FF FF FF FF FF FF FF FF FF FF</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processor</th>
<th>Clock Speed (MHz)</th>
<th>Code Bytes</th>
<th>Execution Time (μsec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSI-11/23</td>
<td>3.33</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>8086</td>
<td>10.00</td>
<td>347</td>
<td>115,669</td>
</tr>
<tr>
<td>68000</td>
<td>10.00</td>
<td>266</td>
<td>33,527</td>
</tr>
<tr>
<td>Z8000</td>
<td>6.00</td>
<td>386</td>
<td>115,500</td>
</tr>
</tbody>
</table>

---

```
procedure QUICKSORT(N,REC,M,WORK)
integer L,R,I,J,K
integer array STACK[O:2*F(N)-1]
character string V

REC[N+1] := oo
L := 1; R := N
do forever
  I := L; J := R+1; V := REC[L]
  do forever
    do I := I+1 until REC[I] >= V end-do
    do J := J-1 until REC[J] <= V end-do
    if J > I
      then swap REC[I] with REC[J]
      else goto end-first
    end-if
  end-do
end-first:

  swap REC[I] with REC[J]
  if both subfile sizes (J-L and R-J) <= M then
    if stack is empty
      then goto end-outer
      else pop L and R from stack
    end-if
    else
      if smaller subfile size <= M
        then set L and R to lower and upper limits of larger subfile
      else push lower and upper limits of larger subfile onto stack
      end-if
  end-if
end-do

else
  push lower and upper limits of larger subfile onto stack
  set L and R to limits of smaller subfile
  end-if
end-do

end-outer:
  do for I from N-1 to 1 in steps of 1
    if REC[I] > REC[I+1] then
      V := REC[I]; J := I+1
      do forever
        REC[J] := REC[J]; J := J+1
        if REC[J] >= V then goto end-last
      end-do
    else
      then swap REC[I] with REC[I+1]
      end-if
    end-do
end-outer

end-last:
  REC[J-1] := V
end-if
end-do
```

---

Fig 6—The Quicksort requires most work from a processor of any of the benchmarks, and it’s typical of a type of application that these devices must frequently serve. The 8086 implementation depends on the 16-byte length of each record and can’t be used for general sorting of records of differing length.
Benchmark K—Bit-matrix transposition

The test data for this benchmark consists of 49 bits in a 7x7 array:

```
0100100
1010111
0010001
1101010
0101000
0000101
1100101
```

The array begins on a word boundary. Timing for the 68000 was hand-computed; the other times come from test runs.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Clock Speed (MHz)</th>
<th>Code Bytes</th>
<th>Execution Time (usec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSI-11/23</td>
<td>3.33</td>
<td>152</td>
<td>1517</td>
</tr>
<tr>
<td>8086</td>
<td>10.00</td>
<td>88</td>
<td>820</td>
</tr>
<tr>
<td>68000</td>
<td>10.00</td>
<td>74</td>
<td>368</td>
</tr>
<tr>
<td>Z8000</td>
<td>6.00</td>
<td>110</td>
<td>646</td>
</tr>
</tbody>
</table>

produce shorter execution times, keep such technological progress in mind if your design project has a long development time or can allow for future upgrading.

"Just the facts, ma'am"

The data in the accompanying boxes summarizes the benchmarking results in terms of code size and execution speed. And the program listings that follow this article illustrate the codings for each processor. As noted, we assume that the manufacturers have done a good job of optimizing the benchmarks; if they don't know how to write code for their own devices, who does?

If there are bugs in the code or ways to improve the coding, EDN would like to know. We have made every effort to check the benchmark programs for correctness and adherence to the specifications. And we thank each of the manufacturers for providing a substantial investment of time and manpower in coding, checking and documentation. We leave any conclusions to you, the reader.

(Ed Note: Some of the benchmarks were not complete at the time this article was prepared. Specifically, the LSI-11/23 Quicksort (Benchmark I) was incomplete, and one of the 8086 timings remained to be determined. We have left the entries for these values blank.)

EDN APRIL 1, 1981 3-91
**BENCHMARK A—68000**

```plaintext
1
2 OPT BRS,FRS
3 * MC68000 EDN BENCHMARK A
4 *
5 * PRIORITY I/O INTERRUPT KERNEL, FOUR PRIORITY LEVELS
6 *
7 * NOTES: 1) FOUR AUTOVECTORS ARE ASSUMED INITIALIZED
8 * TO POINT TO THE FOUR INTERRUPT ENTRY POINTS.
9 * 2) THE MC68000 INTERRUPT SEQUENCE TAKES 4.7
10 * MICROSECONDS WITH AN ASSUMED INTERRUPT
11 * ACKNOWLEDGE BUS CYCLE OF 4 CYCLES.
12 * 3) INTERRUPTS ARE TAKEN ANYWHERE WITHIN THE
13 * MC68000 16 MEGABYTE ADDRESS SPACE.
14 * 4) THE MC68000 PROCESSES INTERRUPTS
15 * IN PRIORITY ORDER WITHOUT REQUIRING THE
16 * SUPPORT OF EXTERNAL CIRCUITY.
17 *
18 * LINES: 8
19 * BYTES: 24
20 *
21 * MC68000L10 BENCHMARK TIME: 33.600 MICROSECONDS
22 *
23 *
24 * INTERRUPT HANDLERS
25 26 00000000 52780018 INTRPT1 ADD $1,COUNTER1 INCREMENT COUNTER FOR INTERRUPT 1
26 27 00000010 4873 RTE RETURN FROM EXCEPTION
28 29 00000006 5278001A INTRPT2 ADD $1,COUNTER2 INCREMENT COUNTER FOR INTERRUPT 2
30 31 00000000 4873 RTE RETURN FROM EXCEPTION
32 33 00000000 5278001C INTRPT3 ADD $1,COUNTER3 INCREMENT COUNTER FOR INTERRUPT 3
34 35 00000010 4873 RTE RETURN FROM EXCEPTION
36 37 38 00000000 5278001E INTRPT4 ADD $1,COUNTER4 INCREMENT COUNTER FOR INTERRUPT 4
38 39 00000000 4873 RTE RETURN FROM EXCEPTION
40 *
41 *
42 *
43 *
44 *
45 END
46
47 ***** TOTAL ERRORS 0-- 0
```

**BENCHMARK A—Z8000**

Example A: I/O Interrupt Kernel, Four Priority Levels

Definitions for interrupt kernel programs:

```plaintext
SYSEM := %5000
SYSTACK := SYSEM + 256
SP := R15
SPOFF := R15
REASON := R1
QEPTR := R2
QENXT := REASON
ADRLEN := 2
JUMP := ADRLEN + 2
EXITOFF := ADRLEN

The four routines that follow are the processing routines for the
four priority levels. VIO is the highest priority routine, VI1 the
lowest. Each of these routines is reached in response to an interrupt on the vectored interrupt (VI) line. Priority
resolution is through a hardware protocol defined as part of the
Z8000 family architecture. Each of the four devices assumed to
be attached to the VI line places its own identifier on the bus
when it interrupts, and this identifier is used by the CPU for
automatic vectoring to the appropriate routine. The addresses
of the routines appear in the program status area (see below).

1 The flag/control word (FCW) value assembled into the
PSA has the vectored interrupt enable (VIE) bit set, so that each processing
routine is interruptible by other vectored interrupt devices.

The hardware interconnection protocol assures that interrupts
come only from higher priority devices.
```

```
```
```
```

Continued on pg 196
BENCHMARK A—Z8000

0010 7B00 IRET
0012 6900 V13: INC VICNT3
0014 0006 ' 
0016 7B00 IRET

!Counters for simulated four priority level processing!
0000 0000 VICNT0: 0
0002 0000 VICNT1: 0
0004 0000 VICNT2: 0
0006 0000 VICNT3: 0

BENCHMARK B—LSI-11/23

1 .TITLE BENCHMARK B
2 .IDENT /OCT.22/
3 .ERASE LC
4
5 ; I/O INTERRUPT KERNEL, FIFO PROCESSING
6 ;
7 ; Services interrupts from four levels, using a FIFO queue.
8 ; Each of the four devices has an interrupt vector set up as follows
9 ;
10 ; = vector address
11 ; =WORD ROUTINE, 340
12 ;
13 ; The vectored interrupt capability of the LSI-11 hardware is used
14 ; here to implicitly identify the device causing the interrupt.
15 ; Each interrupt will vector to a different service routine.
16 ; Hardware will save context (program counter = PC, and processor
17 ; status = PS) at the interrupt.
18 ;
19 000000
20
21 ; Set up a vector for each device.
22 ;
23 000300
24 000302
25 000304
26 000306
27 000308
28 000310
29 000312
30
31 000314
32 000316
33
34 ; Each device operates at priority seven (340 octal in the PS) to disable
35 ; other interrupts.
36 ;
37 ; The queue contains a power of two number of words. It must begin on
38 ; an even multiple of the queue size, such that for all addresses in
39 ; the queue, (address AND queue size) is zero, and ((queue start +
40 ; queue size) AND queue size) is nonzero. QEND points to where the
41 ; next new entry will be made in the queue. QSTART points to where the
42 ; next entry will be removed from the queue. When they point to the
43 ; same place, the queue is empty. We assume the queue never overflows.
44 ;
45 000000
46
47 ; DATA
48 QSIZE = 40 ; sixteen elements
49
50 000000
51 000002
52 000004
53 000006
54 000008
55 000010
56 000012
57
58 000000
59
60 ; Each of the four devices has an interrupt routine as follows:
61 ;
62 ; ROUTINE:
63 ; any immediate processing
64 ; CALL COMMON
65 ; CTR: =WORD 0
66 ;
67 ; CTR is the counter that will be incremented by the FIFO processor.
68 ; In a real example, it would be the first instruction of the
69 ; interrupt service routine.
70 ;
71 000000
72 000002
73 000004
74 000006
75 000008
76 000010
77 000012
78 000014

Continued on pg 200
**BENCHMARK B—68000**

Example B: I/O Interrupt Kernel, FIFO Processing

0000 2DF1 PIFO: EX REASON,SP
0002 93F2 PUSH $SP,QUESTR
0004 6102 LD QUEPTR,QUEIN
0006 000F' LD QUEPTR($ENTOFF),REASON
0008 3212 LD QUEPTR,QUESTR
000A 0002
000C 2121 LD QUEIN,QUEPTR
000E 6F01 LD QUEIN,QUEPTR
0010 000E' TDSEF,FLAG
0012 4C06 TSETB FLAG
0014 003C'
0016 950C JR M1,RESTOR
0018 7C06 LOOP: EI NVI
001A 3121 LD REASON,QUESTR($ENTOFF)
001C 0002
001E 6B10 INC B NVI
0020 0006' DI NVI
0022 7C02 LD QUEPTR($ENTOFF)
0024 2122 CP QUEPTR,QUEPTR
0026 4B02 POP QUEPTR,QUEPTR
0028 000E' JR NE,LOOP
002A 4C06 CLR B FLAG
002C 003C'
002E 97F2 RESTOR: POP QUEPTR,SP
0030 2DF1 EX REASON,SP
0032 7B00 IBET
0034 TOTAL ERRORS $-- $ 

**BENCHMARK B—Z8000**

Example B: I/O Interrupt Kernel, FIFO Processing

0000 2DF1 PIFO: EX REASON,SP
0002 93F2 PUSH $SP,QUESTR
0004 6102 LD QUEPTR,QUEIN
0006 000F' LD QUEPTR($ENTOFF),REASON
0008 3212 LD QUEPTR,QUESTR
000A 0002
000C 2121 LD QUEIN,QUEPTR
000E 6F01 LD QUEIN,QUEPTR
0010 000E' TDSEF,FLAG
0012 4C06 TSETB FLAG
0014 003C'
0016 950C JR M1,RESTOR
0018 7C06 LOOP: EI NVI
001A 3121 LD REASON,QUESTR($ENTOFF)
001C 0002
001E 6B10 INC B NVI
0020 0006' DI NVI
0022 7C02 LD QUEPTR($ENTOFF)
0024 2122 CP QUEPTR,QUEPTR
0026 4B02 POP QUEPTR,QUEPTR
0028 000E' JR NE,LOOP
002A 4C06 CLR B FLAG
002C 003C'
002E 97F2 RESTOR: POP QUEPTR,SP
0030 2DF1 EX REASON,SP
0032 7B00 IBET
0034 TOTAL ERRORS $-- $
BENCHMARK E—68000

1 OPT BRS
2 * MC68000 EDN BENCHMARK E
3 * SUBSTRING CHARACTER SEARCH
4 *
5 * ATTRIBUTES: * 16 MEGABYTE ADDRESSING RANGE
6 * * POSITION INDEPENDENT
7 * * REENTRANT
8 *
9 * INPUT ARGUMENTS:
10 * * D8 - SEARCH PATTERN LENGTH
11 * * A8 - SEARCH PATTERN STRING ADDRESS
12 * * D1 - SEARCHED STRING LENGTH
13 * * A1 - SEARCHED STRING ADDRESS
14 *
15 * OUTPUT:
16 * * D2 - RETURNED MATCHED OFFSET VALUE (-1 IF NO MATCH)
17 * * ALL OTHER REGISTERS ARE TRANSPARENT OVER THIS ROUTINE
18 *
19 * LINES: 18
20 * BYTES: 44
21 *
22 * MC68000L10 BENCHMARK TIME: 244,000 MICROSECONDS
23 *
24 *
25 *
26 *
27 *
28 *
29 *
30 *
31 *
32 *
33 *
34 34 0 0000000 40E718030 34 INDEX MOVEM.L D3/D4/A2/A3,-,(SP)
35 0 0000000 9240 MOVEM.L D8,D1
36 0 0000000 3401 SUB MOVE D1,D2
37 0 0000000 5548 ADJUST COUNT FOR DBRA LOOP
38 0 0000000 1618 SUB PATTERN STRING ADDRESS
39 *
40 *
41 *
42 *
43 *
44 *
45 *
46 *
47 *
48 *
49 *
50 *
51 *
52 *
53 *
54 *
55 *

BENCHMARK E—Z8000

!Example E: Character Search!

!Arguments:
SRCHLENTH := R0 !Length (in bytes) of SRCHRST!
ARGLENTH := R1 !Length (in bytes of ARGCHRG !
SRCHRST := R2 !Address of the string to be searched!
SRCHOFF := R2 !Offset portion of address!
SRCHRARG := R4 !Address of string sought!
ARGOFF := R4 !Offset portion of address!
LOC := R6 !Return arg char position (=0) or -1 (FAILCODE)
FAILCODE := R7 !negative (FAILCODE) if no match!

!Workspace for the routine:
G := RH6 !First char of sought string!
LCT := R7 !Substring counter!
SCH := R8 !Address register used with ARGCHRG!
ARG := R10 !Address register used with SRCHRST!
OFFSAVE := R12 !Reorders original SRCHOFF value!
CT := R13 !Count (bytes in arch string)!
WK1 := R7 !
WK2 := 7

0000 ABFD SEARCH: DEC SP,#2*WK1
0002 1CF9 LDM #SP,WK1,FWK1 !Save register used!
0004 0706 0006 A107 LD LCT,SRCHLENTH !Compute number of substrings!
0008 8377 SUB LCT,ARGLENTH !long enough to match!
BENCHMARK E—Z8000

000A 0A70  INC LCT
000C 0A12C LD OFFSAVE,SRCHOFF  ;Save initial SRCHOFF value!
0002 2046 LDB G,#SRCHARG  ;First char to look for!
000A 0A90  INC ARGOFF  ;Set to compare remainder!
0012 0A810 DEC ARGMOQTH  ;Chars in remainder!

;Check possible substrings from left to right!
0014 B32A GLOOP: CPIRB G,#SRCHSTR,LCT,EQ  ;Match first char!
0016 0766
0018 0E5A JR NZ,FALL  ;No match!
001A 0B14 TEST ARGMOQTH  ;Any more chars in string?!
001C 060B JR Z,MATCH  ;No - already finished!
001A 011D LD CT,ARGMOQTH  ;Set length and!
0020 0A128 LD SRCHSTR  ;string address!
0022 0A14A LD ARG,SRCHARG  ;for block comparison!
0024 B6A0 CPSPRB #ARG,#ARG,CT,NE  ;Look for a mismatch!
0026 0085
0028 0E05 JR NZ,CLODP  ;IStrings match , byte for byte!
002A 0767 TEST LCT  ;No match - try next substring!
002C 0EF3 JR NZ,CLODP  ;if any! ;
002E 2106 FALL: LD LOC,#FAILCODE  ;IMore substrings-fail!
0030 0FFF
0032 B303 JR EXIT!
0034 A126 MATCH: LD LOC,SRCHOFF  ;Match - return index of !
0036 83C6 SUB LOC,OFFSAVE  ;I substring matching initial I!
0038 AB60 DEC LOC  ;I search string!
003A 0C1F1 EXIT1: LDN WK1,ESP,#NW1
003C 0766
003E A6FD INC SP,#2*NW1  ;Restore registers !
0040 0ED8 RET

BENCHMARK F—LSI-11/23

.TITLE BENCHMARK F
IDENT /OCT.22/
.ENABLE LC

; BIT TEST, SET, OR RESET
; 
; Find a bit, check it, change it, bash it, smash it
; 
; Assumes that bits are numbered from 0 from the right-hand end of a word.
; 
; This is the way a PDP-ll views words. Luckily left-to-right ordering
; 
; Was not a benchmark requirement! Arguments are passed on the stack.
; 
; Naturally, performance improvements could be made by passing the arguments
; 
; In registers.
; 
; Stack offsets assume 4 bytes for saved registers:

16 000006  F = 6 ; function code
17 000010  N = 10 ; relative bit number
18 000012  A1 = 12 ; address of bit string
19 000014  HC = 14 ; address of return code word
20 000016  WD&R = 16 ; not used

21 000000  GONTRACE:
23 000000  BTSR:
24 000000  MOV R0, -(SP) ; save registers
25 000022  MOV R1, -(SP) ;
26 000004  MOV 000014  ;
27 000010  MOV 000010  ;
28 000014  MOV 000014  ;
29 000020  MOV 000001  ;
30 000024  MOV 000010  ;
31 000026  MOV 000010  ;
32 000032  MOV 000010  ;
33 000036  MOV 000014  ;
34 000042  MOV 000014  ;
35 000044  MOV 000014  ;
36 000046  MOV 000014  ;
37 000048  MOV 000014  ;
38 000050  MOV 000014  ;
39 000062  MOV 000014  ;
40 000064  MOV 000014  ;
41 000066  MOV 000014  ;
42 000074  MOV 000014  ;
43 000076  MOV 000014  ;
44 000078  MOV 000014  ;
45 000080  MOV 000014  ;
46 000088  MOV 000014  ;
47 000090  MOV 000014  ;
48 000098  MOV 000014  ;

EDN APRIL 1, 1981  3-96
BENCHMARK F—68000

OPT BRS

* MC68000 EDN BENCHMARK F

* BIT ARRAY TEST, SET, AND RESET

* ATTRIBUTES: * 16 MEGABYTE ADDRESSING RANGE
  * POSITION INDEPENDENT
  * REENTRANT

* INPUT:
  * D0 - FUNCTION CODE
  * A0 - BIT ARRAY BASE ADDRESS
  * D1 - BIT SUBSCRIPT INDEX

* OUTPUT:
  * D2 - RETURN CODE

* ALL OTHER REGISTERS ARE TRANSPARENT OVER THIS ROUTINE

* LINES: 15
  * BYTES: 36

* MC68000L1@ BENCHMARK TIME: 78.208 MICROSECONDS

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

\* BIT TEST, SET, RESET SUBROUTINE

\* MOVE.L D1,D2, COPY BIT INDEX OVER
\* LSR.L #3,D2, DIVIDE BY 8 FOR BYTE ADDRESS
\* SUB $2,D0, OFFSET FUNCTION CODE DOWN 2
\* BBR D2, BRANCH IF 2 TO SET
\* SUB $1,D0, DOWN ANOTHER
\* BBR D2, BRANCH IF 3 TO RESET
\* TEST
\* BTST D1,(A0,D2.L), CODE NOT 2 OR 3 - TEST BIT
\* SET
\* SNE D2, SET RETURN CODE SAME AS BIT
\* RTS RETURN TO CALLER
\* TEMP
\* BSET D1,(A0,D2.L), CODE 2 - SET BIT
\* SNE D2, SET RETURN CODE FOR PREVIOUS SETTING
\* RTS RETURN TO CALLER
\* RES
\* BCLR D1,(A0,D2.L), CODE 3 - CLEAR BIT
\* SNE D2, SET RC ONE OR ZERO, SAME AS PREVIOUS
\* RTS RETURN TO CALLER

END

***** TOTAL ERRORS 0--

BENCHMARK F—Z8000

!Example F: Bit array manipulation routines:

!Arguments:
F := RO !Function code - possible values are:
FL := RLO
FH := RHO
TESTCODE := 1 !Test the bit!
SETCODE := 2 !Set the bit to 1!
RESETCODE := 3 !Reset the bit to 0!
N := R1 !Index (from zero) of desired bit!
A1 := R2 !Address of bit array (RR2 for seg)!
AP := R3 !Return arg; set to value of desired bit!

!Workspace for the routine:
BITNUM := RC !Byte offset of desired bit in Al array!
CURBYTE := FH !Temp home of byte containing the bit!

0000 A113 BITMAN: LD BYTNUM,N
0002 B339 SRA BYTNUM,#3
0004 0000 1 I Compute byte offset of |
0006 8132 ADD APOFF,BYTNUM
0008 9200 LDB CURBYTE, A1
000A 3F8E BITB CURBYTE,N
000C 0000 1 I Get the byte!
000E 2010 JR #,NOTSET
0010 BD31 LDK RC, R1
0012 A982 DECH FL,#RESETCODE
0014 9202 RET NE 1 No, exit |
0016 2201 RESB CURBYTE,N 1 Yes, do it |
0018 0000 1
001A 2220 LDB #A1,CURBYTE
001C 9208 RET
001E BD30 NOTSET: LDK RC, R0
0020 AA01 DECH FL,#SETCODE
0022 9202 RET NE 1 No, exit |
0024 2401 SETB CURBYTE,N 1 Yes, do it |
0026 0000 1
0028 2220 LDB #A1,CURBYTE
002A 9208 RET
002C A801 1 I Save changed byte |

Continued on pg 229
BENCHMARK H—68000

Example H: Insertion in a Doubly Linked list!

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADLEN := 2</td>
<td>Number of bytes in an address (4 for segmented operation)</td>
</tr>
<tr>
<td>LISTCB := R12</td>
<td>Address of list control block (RR12 for segmented)</td>
</tr>
<tr>
<td>IFormat of list control block:</td>
<td></td>
</tr>
<tr>
<td>HEADF := 0</td>
<td>IADR of the list &quot;head&quot; (first entry)</td>
</tr>
<tr>
<td>TAILF := ADLEN</td>
<td>IADR of &quot;tail&quot; (last entry)</td>
</tr>
<tr>
<td>AOLEN := TAILF*ADLEN</td>
<td>Number of entries in list!</td>
</tr>
<tr>
<td>NEWENTRY := R10</td>
<td>IADR of entry to be inserted (RR10 for segmented)</td>
</tr>
<tr>
<td>NEWOFF := R10</td>
<td>Offset portion of address!</td>
</tr>
<tr>
<td>IFormat of an entry</td>
<td></td>
</tr>
<tr>
<td>KEYF := 4</td>
<td>IKey portion of entry!</td>
</tr>
<tr>
<td>LKEY := 4</td>
<td>INumber of bytes in a key!</td>
</tr>
<tr>
<td>NEXTF := LKEY</td>
<td>IPPointer to &quot;next&quot; entry!</td>
</tr>
<tr>
<td>PREVF := NEXTF*ADLEN</td>
<td>IPPointer to &quot;previous&quot; entry!</td>
</tr>
</tbody>
</table>

IWorking storage for routine:

<table>
<thead>
<tr>
<th>Register</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY</td>
<td>R80</td>
</tr>
<tr>
<td>PTS2</td>
<td>R2</td>
</tr>
<tr>
<td>NEXTAD</td>
<td>R2</td>
</tr>
<tr>
<td>PREVAD</td>
<td>R3</td>
</tr>
<tr>
<td>NPTRS</td>
<td>ADLEN</td>
</tr>
<tr>
<td>NUM</td>
<td>R4</td>
</tr>
<tr>
<td>WK3</td>
<td>R80</td>
</tr>
<tr>
<td>NW3</td>
<td>NPTRS+3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 A8F9</td>
<td>LISTIN: DEC SP, #2*NW3</td>
</tr>
<tr>
<td>0002 C9F9</td>
<td>LDM SP, WK3, NW3</td>
</tr>
<tr>
<td>0004 0000</td>
<td>ISave registers used!</td>
</tr>
<tr>
<td>0006 14A0</td>
<td>LDL KEY, NEWENTRY</td>
</tr>
<tr>
<td>0008 31C4</td>
<td>LD NUM, LISTCB(#NUMF)</td>
</tr>
<tr>
<td>000A 0004</td>
<td>ICount the new entry!</td>
</tr>
<tr>
<td>000C A940</td>
<td>INC NUM</td>
</tr>
<tr>
<td>000E 33C4</td>
<td>LD LISTCB(#NUMF), NUM</td>
</tr>
<tr>
<td>0010 0004</td>
<td></td>
</tr>
<tr>
<td>0012 BD30</td>
<td>LDX PREVAD, #0</td>
</tr>
<tr>
<td>0014 0B04</td>
<td>CP NUM, #1</td>
</tr>
<tr>
<td>0016 0001</td>
<td>IFirst entry?1</td>
</tr>
<tr>
<td>0018 E055</td>
<td>JR NE, NOTFRST</td>
</tr>
<tr>
<td>001A 0D20</td>
<td>LDX NEXTAD, #0</td>
</tr>
<tr>
<td>001C 03CA</td>
<td>LD LISTCB, NEWENTRY</td>
</tr>
<tr>
<td>001E 33CA</td>
<td>LD LISTCB(#TAILF), NEWENTRY</td>
</tr>
<tr>
<td>0020 0002</td>
<td></td>
</tr>
<tr>
<td>0022 E917</td>
<td>JR UPNEW</td>
</tr>
<tr>
<td>0024 21C2</td>
<td>NOTFRST: LDX NEXTAD, LISTCB</td>
</tr>
<tr>
<td>0026 1020</td>
<td>SCANFL: CPL KEY, #NEXTAD</td>
</tr>
<tr>
<td>0028 E906</td>
<td>JGE, TRNEXT</td>
</tr>
<tr>
<td>002A BD34</td>
<td>TEST PREVAD</td>
</tr>
<tr>
<td>002C E04E</td>
<td>JR NZ, UPMD</td>
</tr>
<tr>
<td>002E 33CA</td>
<td>LD LISTCB, NEWENTRY</td>
</tr>
<tr>
<td>0030 332A</td>
<td>LD NEXTAD(#PREVF), NEWENTRY</td>
</tr>
</tbody>
</table>
| 0032 0006  | IUpdate prev's "next"!
| 0034 E80E  | JR UPNEW |
| 0036 A123  | TRYNEXT: LDX PREVAD, NEXTAD |
| 0038 3132  | LDX NEXTAD, PREVAD(#NEXTF) |
| 003A 0000  | |
| 003C BD24  | TEST NEXTAD |
| 003E E0F3  | JR NZ, SCANFL |
| 0040 333A  | LD PREVAD(#NEXTF), NEWENTRY |
| 0042 0004  | IYes - set prev's "next" |
| 0044 33CA  | LD LISTCB(#TAILF), NEWENTRY |
| 0046 0002  | ISet LISTCB "tail" ptr! |
| 0048 E804  | JR UPNEW |
| 004A 332A  | UPMD: LD NEXTAD(#PREVF), NEWENTRY |
| 004C 0006  | Update new entry's ptrs |
| 0050 0004  | |
| 0052 A943  | UPNEW: INC NEWENTRY, #KEY |
| 0054 1C9A  | LDM #NEWENTRY, PTRS, #NPTRS |
| 0056 0201  | |
| 0058 1CF1  | LDM WK3, #SP, NW3 |
| 005A 0004  | |
| 005C A9F9  | INC SP, #2*NW3 |
| 005E 9E08  | IRestore registers! |

BENCHMARK H—Z8000

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 ABF9</td>
<td>LISTIN: DEC SP, #2*NW3</td>
</tr>
<tr>
<td>0002 1C99</td>
<td>LDM SP, WK3, NW3</td>
</tr>
<tr>
<td>0004 0004</td>
<td>ISave registers used!</td>
</tr>
<tr>
<td>0006 14A0</td>
<td>LDL KEY, NEWENTRY</td>
</tr>
<tr>
<td>0008 31C4</td>
<td>LD NUM, LISTCB(#NUMF)</td>
</tr>
<tr>
<td>000A 0004</td>
<td>ICount the new entry!</td>
</tr>
<tr>
<td>000C A940</td>
<td>INC NUM</td>
</tr>
<tr>
<td>000E 33C4</td>
<td>LD LISTCB(#NUMF), NUM</td>
</tr>
<tr>
<td>0010 0004</td>
<td></td>
</tr>
<tr>
<td>0012 BD30</td>
<td>LDX PREVAD, #0</td>
</tr>
<tr>
<td>0014 0B04</td>
<td>CP NUM, #1</td>
</tr>
<tr>
<td>0016 0001</td>
<td>IFirst entry?1</td>
</tr>
<tr>
<td>0018 E055</td>
<td>JR NE, NOTFRST</td>
</tr>
<tr>
<td>001A 0D20</td>
<td>LDX NEXTAD, #0</td>
</tr>
<tr>
<td>001C 03CA</td>
<td>LD LISTCB, NEWENTRY</td>
</tr>
<tr>
<td>001E 33CA</td>
<td>LD LISTCB(#TAILF), NEWENTRY</td>
</tr>
<tr>
<td>0020 0002</td>
<td></td>
</tr>
<tr>
<td>0022 E917</td>
<td>JR UPNEW</td>
</tr>
<tr>
<td>0024 21C2</td>
<td>NOTFRST: LDX NEXTAD, LISTCB</td>
</tr>
<tr>
<td>0026 1020</td>
<td>SCANFL: CPL KEY, #NEXTAD</td>
</tr>
<tr>
<td>0028 E906</td>
<td>JGE, TRNEXT</td>
</tr>
<tr>
<td>002A BD34</td>
<td>TEST PREVAD</td>
</tr>
<tr>
<td>002C E04E</td>
<td>JR NZ, UPMD</td>
</tr>
<tr>
<td>002E 33CA</td>
<td>LD LISTCB, NEWENTRY</td>
</tr>
<tr>
<td>0030 332A</td>
<td>LD NEXTAD(#PREVF), NEWENTRY</td>
</tr>
</tbody>
</table>
| 0032 0006  | IUpdate prev's "next"!
| 0034 E80E  | JR UPNEW |
| 0036 A123  | TRYNEXT: LDX PREVAD, NEXTAD |
| 0038 3132  | LDX NEXTAD, PREVAD(#NEXTF) |
| 003A 0000  | |
| 003C BD24  | TEST NEXTAD |
| 003E E0F3  | JR NZ, SCANFL |
| 0040 333A  | LD PREVAD(#NEXTF), NEWENTRY |
| 0042 0004  | IYes - set prev's "next" |
| 0044 33CA  | LD LISTCB(#TAILF), NEWENTRY |
| 0046 0002  | ISet LISTCB "tail" ptr! |
| 0048 E804  | JR UPNEW |
| 004A 332A  | UPMD: LD NEXTAD(#PREVF), NEWENTRY |
| 004C 0006  | Update new entry's ptrs |
| 0050 0004  | |
| 0052 A943  | UPNEW: INC NEWENTRY, #KEY |
| 0054 1C9A  | LDM #NEWENTRY, PTRS, #NPTRS |
| 0056 0201  | |
| 0058 1CF1  | LDM WK3, #SP, NW3 |
| 005A 0004  | |
| 005C A9F9  | INC SP, #2*NW3 |
| 005E 9E08  | IRestore registers! |

3-98 EDN APRIL 1, 1981
**BENCHMARK I—68000**

```
96  # 00000AA 40D3000F
97  # 000000E 40D00000
98  # 0000002 22B9
99  # 0000005 24B8
100 # 0000009 9282
101 # 000000A 94B9
102 # 0000009 84B8
103 # 000000C 62C6
104 # 000000F 826E
105 # 0000008 62DB
106 # 0000002 4CD70000
107 # 0000006 66000F6C

LOOP CONTROL
TEMP
COUNTER AND SWAP REGISTER
WORK REGISTERS
"V· SAVE

Example I: Quicksort/Insertion

Arguments

N := R0  Number of records
M := R1  Changeover point
REC := R2  Array base
RECOFF := R3
RECOED := R5

Working registers

SCL := R0  Scratch borrowed from argument registers
SCR1 := R0
SCR2 := R1
BIGM := SCL
ADR := RH4; ADRHH := RH4; ADRHL := RH4; ADR := RH5
J := R8; JHI := R6; ILO := R7
L := R10; LHI := R8; JLO := R9
U := R12; UHI := R12; ULO := R13

Example I: Quicksort/Insertion Sort

BENCHMARK I—Z8000

Arguments

N := R0  Number of records
M := R1  Changeover point
REC := R2  Array base
RECOFF := R3
RECOED := R5

Working registers

SCL := R0  Scratch borrowed from argument registers
SCR1 := R0
SCR2 := R1
BIGM := SCL
ADR := RH4; ADRHH := RH4; ADRHL := RH4; ADR := RH5
J := R8; JHI := R6; ILO := R7
L := R10; LHI := R8; JLO := R9
U := R12; UHI := R12; ULO := R13

ITAD := ADRL  Address of I-item

Example I: Quicksort/Insertion Sort

"
BENCHMARK I—Z8000

JEAD := LLO  IAddress of J-item I
PIVOT := SCRL
PIVHI := SCR1
PIVLO := SCR2

ITemporary registers for item moving I
DEST := LLO
SRCE := ADRL

Other constants I
ESIZE := 16  IBytes per record I
KEYOFF := 3  IIndex in record of first byte of key I
KEYBITES := 7  IBytes per key I

0000 ABFF  SORT:  DEC SP,#16
0002 ABFD  DEC SP,#12
0004 1C99  LDM @SP,RO,#14  ISave all registers I
0006 0000  LDI ULO,N  INumber of records I
0008 A100  EXTS U
000A A10A  MUL U,ESIZE  IMult by size of records I
000E 0010  LDI L,#0  IZero lower limit to start I
0010 140A  LDL L,#ESIZE
0012 0000
0014 0000
0016 01FC  PUSHL @SP,U
0018 1900  MULI BIGH,ESIZE  IAdjust cutoff for record size
001A 0010
001C 91F0
001E D7F7  CALR QUICK
0020 95F0  POPL BIGH,SP
0022 95F6  POPI BHIGH
0024 1206  SUBL I,ESIZE
0026 0000
0028 0010
002A 9464  INSORT:  LDL ADR,I
002C 160A  ADDL ADR,ESIZE
002E 0000
0030 0010
0032 8135  ICALR ADCOMP1
0034 1604  ADDL ADR,RECOFF
0036 0000
0038 0003
003A 9440  LDL PIVOT,ADR  IPIVOT is adr of key of A(I+1) I
003C D7F0  CALR CPPI
003E EF2F  IF A(I+1)>A(I), end block I
0040 ABFF  DEC SP,ESIZE
0042 9008  CLR PIVHI
0044 A1P1  LDL PIVLO,SP  IPIVOT has adr of V on stack I
0046 1600  ADDL PIVOT,KEYOFF  IPIVOT points to key in V I
0048 0000
004A 0003
004C A1FB  LDL DEST,SP
004E 9464  ICALR ADCOMP1  IADR is source address I
0050 8135  ADDL ADR,RECOFF
0052 8DA8  LDL @10,ESIZE/2
0054 B851  LDIE #DEST,%ROE,110  ISave a(I) on stack I
0056 C800  LDL J,I
0058 9468  ADDL J,ESIZE  IJ = I + 1 I
005A 1608
005C 0000
005E 0010
0060 948A  AGR2:  LDL ADR,J
0062 1204  SUBL ADR,ESIZE  IADR = J - 1 I
0064 0000
0066 0010
0068 8135  ICALR ADCOMP1
006A 944A  LDL L,ADR  IL = address of A(J-1) I
006C 948A  ICALR ADCOMP1
006E 8135  ADDL ADR,RECOFF
0070 8DA8  LDL @10,ESIZE/2
0072 B851  LDIE #DEST,%ROE,110  IA(J-1) = A(J) I
0074 C800  ADDL J,ESIZE  IJ = J + 1 I
0076 1608
0078 0000
007A 0010
007C D9F8  CALR CPPI
007E E401  JB OV,ENDLAST
0080 E7FF  ENDL1:LDLD ADR,J
0082 948A  SUBL ADR,ESIZE  IADR = J - 1 I
0084 1204
0086 0000
0088 0010
008A 8135  ICALR ADCOMP1
008C 944A  LDL L,ADR
008E 940A  LDL ADR,PIVOT
0090 1204  SUBL ADR,KEYOFF  IADR = address of V again I
0092 0000
0094 0003
0096 8DA8  LDL @10,ESIZE/2
0098 B851  LDIE #DEST,%ROE,110  IA(J-1) = V I

3-100  EDN APRIL 1, 1981
BENCHMARK I—Z8000

009A 0A0B INC SP,#ESIZE
009C 40FF SUBL I,#ESIZE
009E 1206 END1: SUBL I,#ESIZE
00A0 0000
00A2 0010
00A4 9C68 TESTL I
00A6 EEC1 JR NZ,INSORT
00AB 1C01 LDM RO,#SP,#14
00AA 000D INC SP,#16
00AC 49FF INC SP,#12
00BE 0E08 I!restore registers I
00B0 0E08 RET

I!Subroutine Quicksort — after C. A. R. Hoare
CALL QUICK with BASE = array address
U = offset of upper limit
L = offset of lower limit
Semi-sorts elements at offsets between L and U (inclusive).
The 23-bit integers L and U are in the range 0 to 8,388,607.

00B2 94C4 QUICK: LDL ADR,U
00B4 92A4 SUBL ADR,L Icompute subfile size IC
00B6 9004 CPL ADR,#BIGH IReturn if subfile is <= H long
00B8 9F02 RET LE IReturn if subfile is <= H long
00BA 91F0 "Partition array segment between offsets L and U (inclusive)
around a pivot element with index J. Returns the ranges:
(J,J-1) in L,U
(J+1,U) in I,J

00BC 94A4 PART: LDL ADR,L IADRL = L IC
00BE 8135 IADRP = actual address of a(L) IC
00C0 1604 ADD ADR,#RECOFF IC
00C2 0000 ADDL ADR,#KEYOFF Iadd in offset of key within record IC
00C4 0003 ADDL ADR,#ADCPM IC
00C6 9410 LDL PIVOT,ADR IPIVOT = actual address of pivot IC
00C8 94A6 LDL I,L IC
00CA 94A6 LDL J,U IC
00CC 1608 ADDL J,#ESIZE IJ = J+1 IC
00CE 0000
00D0 0010
00D2 1F1C PUSHL #SP,L IC
00D6 FD07 "LDL PIVOT,ADR IC
00DB 91FA "LDL I,L IC
00DE 91FC "LDL J,U IC
00DF FD07 "PUSHL #SP,U IC
00E8 FD01 "LDL ADR,J IC
00EB 94B4 "IADRP = actual address of a(J) IC
00EC 8135 "ADD ADR,#RECOFF IC
00F0 94A4 LDL L,ADR IC
00F2 9068 CPL J,I IC
00F4 2202 "JR LE,MOPIV IC
00F6 9D05 "CALR EXCHJ IC
00F8 6F07 "JR LPI IC
00FA 9D04 "CALR EXCHUP IC
00FB 95FC "POPL U,#SP IC
00FC 95FA "POPL L,#SP IC
00FD 95B7 "POPL J,#SP IC
00FE 91FC "POPL L,#SP IC
00F0 9486 LDL I,J IPut J into RRA IC
00F2 946C LDL U,I IC
00F4 120C SUBL U,#ESIZE IC
00F6 0000 ADDL I,#ESIZE IC
00F8 0010 ADDL I,#ESIZE IC
00FA 1606 ADDL I,#ESIZE IC
00FC 0000 ADDL I,#ESIZE IC
00FE 0010 ADDL I,#ESIZE IC

I!Put shorter range in L,U, longer in I,J IC
SHORT: LDL SCRL,J IC
ISCR = U-L for first range IC
0100 94B0 LDL SCRL,I IC
0102 94D0 LDL SCRL,CL IC
0104 94B0 LDL SCRL,U IC
0106 94D0 LDL SCRL,L IC
0108 94A0 ADDL SCRL,CL IC
010A 94B0 ADDL SCRL,AR IC
010C 9204 JS LE,91 IC
010E 2467 EX IHI,LHI IC
0110 AD67 EX ILO,LLO IC
0112 ADC8 EX JHI,ULI IC
0114 ADD9 EX JLO,ULU IC
0116 95F0 POPL BIGH,#SP IC
0118 ABFF "DEC SPOFF,#8 IC
011A 1C0F LDM #SP,IHI,#4 IC
011C D603 CALR QUICK IC
011E D037 LDM LHI,#SP,#4 IC
0120 1C0F "CALR QUICK IC
0122 A03 IC
0124 4F07 "INC SPOFF,#8 IC
0126 D038 "CALR QUICK IC
0128 B038 "CALR QUICK IC
012A 1606 UPI: ADDL I,#ESIZE IC
012C 0000 ADDL I,#ESIZE IC
012E 0010 ADDL I,#ESIZE IC
0130 0000 ADDL I,#ESIZE IC

I!Subroutines for moving I and J IC
CALL UPI: Increment I until a(I) >= pivot value IC
CALL DOWNJ: Decrement J until a(J) <= pivot value IC

0132 101
BENCHMARK I—Z8000

012C 0000  CALL CPPI  (Compare pivot value with a(I))
012E 0010  RET OV  (OV = 1 says pivot = a(I))
0130 0077  RETURN  (Return if pivot value <= a(I))
0132 9E04  SUB J, #ESIZE  (Decrement J)
0134 9B07  JR UPI
0136 009F  IPivot and exchange subroutines
0138 1D08  CALR CPPI  (Compare pivot value with a(I))
013A 0000  RET UGE  (Return if pivot >= a(J))
013C 0010  JR DOWJ
013E 9E04  CALL CPPI  (Compare pivot value with a(I)). Set FLAGS.
0140 9B07  RETURN  (Return if pivot value <= a(I))
0142 9E04  CALL EXCHJ - exchange a(I) and pivot values
0144 9B07  RETURN  (Return if pivot value <= a(J))
0146 8E01  LDADR, I  (actual address of a(J) for exchange routines)
0148 946A  CPPI :  LDADR, I  (actual address of a(J) for exchange routines)
014A 8135  ADDADR, RECOFF  (Set flags)
014C 1604  ADDADR, #KEYOFF  (Set flags)
014E 0000  0150 0003  0152 940A  LLADR, PIVOT  (IADR = address of a(J))
0154 BDC7  LDKH, #KEYTIE  (Number of bytes in key)
0156 BA56  CPBIRD, #DEST, #SRC, #UI, #E
0158 0CE8  OBE  (Exchange)
015A 9D08  RET
015C 946A  EXCHJ :  LDADR, J  (Exchange)
015E 8135  ADDADR, RECOFF  (Set flags)
0160 E604  EXCH :  LDR, I  (Exchange)
0162 9404  EXCHJ :  LDADR, PIVOT  (Exchange)
0164 1204  SUBLADR, #KEYOFF  (Exchange)
0166 0000  0168 0003  016A BDC8  EXCH :  LDKH, #ESIZE/2  (Record word count)
016C 215D  EXUO, #TID  (Pick up pivot or a(I) value)
016E 20BD  EX #TID, UO  (Exchange with a(I))
0170 2F5D  LD #TID, UO  (a(I) or pivot = a(J))
0172 A951  INCADR, #2  (Exchange)
0174 A9B1  INC LL0, #2  (And repeat for whole record)
0176 FC86  DJNZ UHI, EXLOOP  (And repeat for whole record)
0178 9D08  RET
017A A4C4  ADComp :  EXB ADRRH, ADRRL  (Add offset of REC to low index)
017C 8135  ADDADR, RECOFF  (Add seg of REC (with C) to high)
017E B424  ADDADR, #DESEG  (Add seg of REC (with C) to high)
0180 9E08  RET

BENCHMARK K—LSI-11/23

. TITL BENCHMARK K
. IDENT /OCT.23/  ; BOOLEAN MATRIX TRANSPOSE
. EMUBL LC  ; Transpose a tightly-packed bit matrix
 ; Arguments are passed on the stack.
 ; Offsets assume 14(k) bytes used for saving registers on stack.
 ;
 ; size of matrix
 ;
 ;
12 000016  N = 16
13 000020  A1 = 20
14 000022  A2 = 22
15 000000
16 000000  ONTRACE:
17 000200  BMII:  ; save registers
18 000016  MOV R0, -(SP)
19 000008  MOV R1, -(SP)
20 000004  MOV R2, -(SP)
21 000006  MOV R3, -(SP)
22 000010  MOV R4, -(SP)
23 000012  MOV R5, -(SP)
24 000014  MOV R6, -(SP)
25 000015  MOV R7, -(SP)
26 000016  MOV R8, -(SP)
27 000018

Continued on pg 256
BENCHMARK K—68000

Example K: Boolean Matrix Transpose

Arguments:
NX := R0 Number of Columns of Matrix
AJ := R1 Base Address of Matrix
ALX := R2 Address of First Word of Matrix

Working storage for the routine:
WK5 := R4
NWS := 9
IJBYTE := RH4 Byte containing a(I,J)
JBYTE := RL4 Byte containing a(J,I)
TWOBITS := RB5 Holds both bit values
IJPTR := R6 Bit number of all(I,J)
JIPTR := R7 Bit number of all(J,I)
JIBYTE := R8 Address of IJ byte
JIX := R10 Address of JI byte
JIBX := R11 JIBX for outer loop
LPCNT := R12 Counter for outer loop

Long registers for one-step loads:
BXL := R8
OFFL := R9

Code for Boolean matrix transpose:

BENCHMARK K—Z8000

Example K: Boolean Matrix Transpose

Arguments:
NX := R0 Dimension of Matrix
A2 := R1 Bit (0<=A2<=15) at which matrix begins in A1
ALX := R2 Address of first word of Matrix

Working storage for the routine:
WK5 := R4
NWS := 9
IJBYTE := RH4 Byte containing a(I,J)
JBYTE := RL4 Byte containing a(J,I)
TWOBITS := RB5 Holds both bit values
IJPTR := R6 Bit number of all(I,J)
JIPTR := R7 Bit number of all(J,I)
JIBYTE := R8 Address of IJ byte
JIX := R10 Address of JI byte
JIBX := R11 JIBX for outer loop
LPCNT := R12 Counter for outer loop

Long registers for one-step loads:
BXL := R8
OFFL := R9

Code for Boolean matrix transpose:

K-Z8000
Some fundamental constraints on microprocessor peripheral families have always existed, but some of the more severe constraints in the present 16-bit environment will be worse in future 32-bit environments. One of these restrictions is the number of signal lines available—usually corresponding to the number of pins on a package. Present packaging technology for mass-produced parts allows up to 64 pins, which is sufficient for a 16-bit microprocessor with an unmultiplexed address/data bus or a 32-bit microprocessor with a multiplexed address/data bus. Unfortunately, control of these wide buses uses most of the pins available with current packaging, so any device controlling the bus cannot have a wide, independent data path.

The key word here is "independent." It is certainly possible to design a device that could operate a local bus and, when necessary, switch modes to control a global bus. This mode of operation for multiple processor-type devices is inferior for several reasons. First, when the buses are linked, other processes experience longer delays in being serviced. Second, an architecture that allows multiple-processor devices access to most memory in the system is a difficult one in which to assure data and system integrity. A third difficulty is simply the number of devices necessary to link the buses. Typical implementations require six to eight packages.

A significant observation is that the only commercially available I/O devices that incorporate a DMA-type function are serial input/output devices and CRT controllers. Only these applications allow enough pins to properly implement the DMA function.

Fortunately, the same technology that enables the integration of 16- and 32-bit microprocessors also allows the integration of considerable intelligence and some buffer memory in the peripheral device. This is a very powerful combination, especially in conjunction with highly integrated CPU/DMA combinations, and can be used to link multiple local buses to a main system bus at high speed and with little overhead.

Local buses are, in general, a very effective way to improve overall system performance. They allow significant parallel processing to occur and can improve system reliability by partitioning the tasks to make interference between processes less likely. Many of the problems with linking multiple buses can be avoided by adding buffer memory between the buses. In many of the new-generation I/O devices, this buffer memory can be included on the integrated circuit itself.

An example of the power of these techniques is the construction of a high-speed parallel/serial front-end processor for a high-end microcomputer system (Figure 1).

The key element in this system is the Z8038 FIFO (FIFO Input/Output) device. This is a 128x8 FIFO buffer that has the necessary intelligence and flexibility to interface to a wide variety of microprocessors. It also has the ability to interrupt under a variety of conditions and can bypass the data FIFO by a separate path to pass control and status information from one processor to another.

Information is passed from one processor to the other on a message basis. A typical transfer begins with the main system processor sending a control byte through the FIFO to the local processor via the bypass register. This control communication typically includes information about the data block length, the intended destination, and any other relevant parameters. At the same time, the main system DMA can be set up to begin transferring data into the FIFO. Either of the two DMA controllers in this system can be eliminated with little loss in performance if the CPU has block memory-to-I/O move instructions available, as in the Z80 or Z8000. After initial setup of the FIFO, the main system DMA is activated and quickly
fills the FIO's data buffer, if the local system DMA has not yet been activated. This is of little consequence, since the main system DMA will simply stop transfers when the RDY signal from the FIO goes inactive. Similarly, if a block move instruction is being used instead of a DMA, the FIO provides an "interrupt-on-full" interrupt, which allows the CPU to do other tasks until next interrupted by the FIO. This second interrupt occurs only when the contents of the FIO have been emptied to a predetermined programmable level.

Similarly, on the local bus side of the FIO, the DMA will be active only when there is data remaining in the FIO. To reduce the number of bus request cycles (or interrupts in the case of a block move instruction), the FIO can be programmed to request service from the local DMA only when the FIFO contains more than a certain programmable number of bytes. It will then transfer until the FIFO is empty and continue this burst cycling until the end of the block.

The combination of the block move instructions and the FIO is more powerful than the replacement of the DMA function. Unlike the DMA, which, by requesting the system bus, places itself at a higher priority than any interrupt in the system, the block move instructions can be interrupted. This means that a high-priority interrupt in either the local system or the main system can be serviced immediately, even though the CPU is involved in a very high speed transfer of data through the FIO. If the interrupt routine is short, the other system may not even notice that the FIO was not being serviced for a short interval. If the interrupt is longer, the fact that the FIFO may go empty is of little consequence. An interrupt on empty or an inactive RDY line will serve to temporarily suspend service of the FIO at the local end.

The FIO is sufficiently flexible to interface in four distinct applications:
- To a multiplexed address/data bus microprocessor.
- To an unmultiplexed address/data bus microprocessor.
- With handshake lines to most types of parallel-interface I/O devices.
- As a "high byte portion" of a 16- or 32-bit link between buses.

Figure 1 also shows the use of the FIO in a handshake application. One of the principal advantages of the FIO in this configuration is its ability to decrease interrupt handling overhead by more than two orders of magnitude, compared to the typical interrupt handling with a parallel I/O device. For example, if interfaced to a line printer, the CPU would be interrupted once per line rather than once per character. Another capability of the FIO is its ability to recognize special characters (or bits in a character). It can interrupt or stop DMA transfers when a special character comes through the FIFO, such as End of File.

![Diagram](Image URL)
The other device shown in the example is the Z8030 Z-SCC (Serial Communications Controller). This device can interface to nearly any type of serial device at up to a speed of 1 million bits per second. This includes all popular asynchronous formats and IBM Bisync (including Transparent mode), as well as the newer protocols such as X.21, X.25, SDLC, and HDLC. In its various modes, the Z-SCC can generate and check the two most popular CRCs (Cyclic Redundancy Codes). It also provides parity generation and checking and handles various lengths of characters.

One major advantage of the Z-SCC over previous serial communications devices is its ability to do all clock recovery and generation for most types of encoding. Specifically, it can encode and decode NRZI as well as FM encoded data with transitions being interpreted as either 1s or 0s. It can also recover both clock and data from Manchester encoded data.

In addition to its clock recovery capabilities, the Z-SCC has two timers for independent baud rate generation in each full duplex channel. The timing sources can be the Z-SCC control clock, an external clock source, or the output of either of the on-chip crystal oscillators. This extreme flexibility in timing allows complete on-chip local loopback testing. An Auto-Echo mode is also provided for modem and link testing.

In keeping with the trend of increased buffer memory, the Z-SCC has sufficient on-board buffering (four characters in the receiver) to allow time for interrupt response even at relatively high data rates. If DMA control becomes necessary for even faster data transfer, this can be accomplished in a full duplex manner in both channels.
Interfacing to the Z6132 Intelligent Memory

Application Note

April 1981

Introduction

In memory applications where the requirements for byte-wide buffer storage are modest (2K to 32K bytes), the Z6132 offers a new concept in intelligent memory. The Z6132 features 4K bytes of RAM in a byte-wide, 28-pin package that conforms to the 2716/2732 JEDEC standard.

This application note discusses the basic features and operating modes of the Z6132, with application examples given for each of Zilog’s microprocessors. In addition to a discussion of the interface requirements for the Z8™, Z80® and Z8000™ CPUs, an application example describes the design requirements for interchanging the Z6132 with 2716/2732-type EPROMs. Each interface design includes logic and timing diagrams. Other Zilog documents that might be useful are referenced throughout the application note.

The application note is divided into four sections. The first section provides a general description of the Z6132 along with functional descriptions of each available type of memory operation. The self-refresh operation is discussed with the various refresh options available with the Z6132. The second section begins the application examples by providing interface circuitry for the Z80A CPU. The third and fourth sections provide interface circuitry and timing for the Z8002 and Z8 microprocessors. The section that discusses the Z8 memory interface also treats the design criteria for interchanging the Z6132 with either 2716- or 2732-type EPROMs.

General Description of the Z6132

The Zilog Z6132 is a +5 V, intelligent, MOS dynamic RAM organized into 4096 8-bit words. The Z6132 uses high-performance, depletion-load, double-poly, n-channel, silicon-gate MOS technology with a mixture of static and dynamic circuitry that provides a small memory cell and low power consumption. Internally, the Z6132 uses dynamic storage cells, but externally, the Z6132 functions as a static RAM because it controls and performs its own refresh. This eliminates the need for external refresh support circuitry and combines the convenience of a static RAM with the high density and low power consumption normally associated with dynamic RAMs.

The Z6132 is particularly well suited for microprocessor and minicomputer applications where its byte-wide organization, self-refresh, and single power-supply voltage result in a reduced parts count and a simplified design. The Z6132 supports both multiplexed and non-multiplexed address and data lines using the control signals Address Strobe (AS) and Data Strobe (DS) to latch address and data internal to the memory chip. The circuit is packaged in an industry-standard, 28-pin DIP and is pin compatible with the proposed JEDEC standard. The Z6132 conforms with the Z-BUS specification used by the new generation of Zilog microprocessors, the Z8 and Z8000.

The Z6132 4K x 8 quasi-static RAM is organized as two separate memory-bit blocks. Each block has 128 sense amplifiers with 64 rows of memory bits on each side. Both blocks have separate row address buffers and decoders. The two sets of row address decoders are addressed either by the address inputs A1–A7 or by the internal 7-bit refresh counter. The least significant address input (A0) selects one of the two blocks for external access. While the selected block performs a read or write operation, the other memory block uses the refresh counter address to refresh one row. Details of the self-refresh mechanism are discussed in the next section.

A memory cycle starts when the rising edge of Address Clock (AC) clocks in Chip Select (CS), A0, and Write Enable (WE). If the chip is not selected (CS is High), all other inputs are ignored until the next rising edge of AC. If the chip is selected (CS is Low), the 12 address bits and the Write Enable bit are
clocked into their respective internal registers. The block addressed by \( A_1-A_{11} \) is determined by \( A_0 \); the other block is refreshed by the 7-bit refresh counter.

The Chip Select and address inputs must be held valid for only a short time after the rising edge of AC. This supports the multiplexing of address and data and allows enough setup time for the multiplexed data lines to settle with respect to the input control signal Data Strobe.

A read cycle is initiated by the rising edge of AC while CS is Low and WE is High. A Low on the DS input activates the data outputs after a specified delay. During a read operation, DS is only a static Output Enable signal.

Write cycle is initiated by the rising edge of AC while both CS and WE are Low. The WE input is checked again on the falling edge of DS. If WE is still Low, the falling edge of DS strobes the data on the D\(_0\)-D\(_7\) inputs into the addressed memory location. Data must be valid for only a short hold time after the falling edge of DS.

**Figure 1. Block Diagram**

Self-Refresh Operation

The Z6132 stores data in a single-transistor dynamic cells that must be refreshed at least every 2 ms. Each of the two memory blocks contains 16,384 cells and requires 128 refresh cycles to completely refresh the array. The Z6132 operates in one of two user-selectable self-refresh modes, each satisfying the refresh time requirements. On the basis of the available memory cycle time, the user can decide to use either the Long Cycle-Time Refresh mode or the Short Cycle-Time Refresh mode. The Long Cycle-Time Refresh mode is the simplest self-refresh mode and is enabled by permanently grounding the BUSY output pin of the Z6132. Every memory cycle in this mode consists of a memory operation followed by a refresh operation on both blocks, after which the refresh counter is incremented. Internally, the complete cycle consists of a four-phase sequence:

1. Memory read, write, or write inhibit
2. Precharge
3. Refresh
4. Precharge

These internal operations are automatic and transparent to the user. When the chip is not selected (CS is High when AC goes High), the first two phases are omitted. There are two important requirements: the memory cycle...
Self-Refresh Operation (Continued)

times must always be longer than the TC (minimum memory cycle time) value specified when BUSY is Low, and there must be at least 128 Address Clocks in any 2 ms period.

The Long Cycle-Time Refresh mode is most practical for microprocessor applications where the read and write cycle times are in the range of 650-750 ns. The Short Cycle-Time Refresh mode is a more sophisticated self-refresh scheme that is activated by pulling the BUSY output pin High through a pullup resistor (typically 1 kΩ) to VCC. The BUSY outputs of several Z6132 chips can be OR-wired together. In this mode, the Z6132 always performs a refresh operation on the memory block that is not being addressed from the outside.

If the chip is selected (CS is Low when AC goes High), the refresh counter refreshes the block that is not addressed by A0. The refresh counter is incremented after both an even and an odd address have occurred. This self-refresh scheme takes advantage of the sequential nature of most memory addressing. If the chip is deselected (CS is High when AC goes High), both blocks are refreshed and the refresh counter is incremented after every cycle. Hence, the addressing of PROM or I/O can also be used to refresh the Z6132 by allowing it to receive Address Clocks without Chip Select.

Under normal conditions, the deselected and odd/even self-refresh mechanisms step through 128 refresh addresses in less than 2 ms. To guarantee proper refresh operation, even in the exceptional case of the memory being continually selected and addressed by a long string of all even or all odd addresses, a built-in cycle counter activates the BUSY output and requests a lengthened memory cycle to append a refresh operation. This internal cycle counter is reset whenever the refresh counter is incremented. The cycle counter then counts memory cycles and activates the BUSY output when it reaches a count of 17.

BUSY is fed into the WAIT input of most microprocessors and is a request to the CPU for a longer memory cycle. The BUSY line is held Low by the Z6132 until the refresh cycle has started. BUSY becomes active only when the Z6132 has been selected and addressed with all odd or all even addresses for 17 consecutive Address Clocks.

Interfacing the Z6132 to the Z80A CPU

The Z6132 was designed to interface with Z-BUS™-compatible microprocessors such as the Z8 and Z8000. Although the Z80 does not directly produce Z-BUS-compatible memory signals, only three commonly available integrated circuits are required to interface the Z6132 with the Z80A CPU. The interface logic, circuit description, and timing diagrams for each important processor cycle are discussed later. Further information on the Z6132 and Z80A CPU can be obtained from the Z6132 Product Specification (document number 00-2028-A) and the Z80B CPU AC Characteristics (document number 00-2005-A).

The M1 or opcode fetch cycle of the Z80A CPU represents the shortest memory cycle and must be given careful consideration when designing memory interface logic. Figure 2 shows the Z80A CPU M1 cycle in detail along with worst-case delay timings for the important control signals. The maximum access time allowed for an opcode fetch (under ideal conditions) is 500 ns in clock cycles T1 and T2. Considering worst-case Z80A CPU data setup time (35 ns in T2) and worst-case opcode address stable time (110 ns in T1), the maximum access time available for a memory fetch is reduced to 355 ns.

To keep the interface logic for the Z6132 to a minimum and still use commonly available parts, the Z6132-5 (300 ns access time) is exemplified. Timing edges provided by the Z80A CPU clock are used to activate the Z6132 Address Clock (signal AC shown in Figures 2, 3, 4, and 5). Figure 7 shows the logic for the Z80A-to-Z6132 interface. The 74S00 NAND gate has a maximum delay of 5 ns, the 74LS04 inverter has a maximum delay of 15 ns, and the 74S74 has a maximum clock to output delay of 9 ns. The clear-to-Q output Low delay is 8 ns for the 74S74. These numbers are displayed in the timing diagrams for the Z6132 control signals CS, AC, DS, and WE (Figures 2-6).

The following description of a memory fetch cycle illustrates how each of the important Z6132 timing parameters is met. The M1 cycle begins with the activation of Z80A CPU control signal M1 in clock cycle T1. Since the maximum delay for M1 is 100 ns (Figure 2) and the
maximum delay from the rising edge of $T_1$ until addresses are stable is 110 ns, the control path that gates $M\bar{I}$ and CLK to clear the 74S74 flip-flop is used to force AC High.

The delay of 33 ns shown in Figure 2 for AC from the falling edge of $T_1$ was derived from the collective delays of the 74LS04 (15 ns), the 74S00 (5 ns), the 74S74 clear (8 ns), and the final 74S00 gate (5 ns). Thus, under the worst conditions possible, a memory cycle begins with the rising edge of AC 158 ns after the rising edge of clock cycle $T_1$.

As a reminder, the $M_1$ machine cycle is a 2-clock-cycle instruction fetch, which requires the data fetched to meet the specified setup time (35 ns) before the rising edge of clock cycle $T_3$. With 35 ns required for worst-case data setup time, the remaining time in $T_1$ and $T_2$ for memory access is:

$$500 \text{ ns} - (158 \text{ ns} + 35 \text{ ns}) = 307 \text{ ns}$$

This allows the use of 300 ns access time RAMs even under worst-case conditions.

The Z6132-5 has a guaranteed access time of

![Figure 2. Z80A Opcode Fetch Cycle Timing](image-url)
Interfacing the Z6132 to the Z80A CPU (Continued)

300 ns and is recommended for use with the Z80A CPU to simplify interface circuitry. This mode takes advantage of the self-refresh feature of the Z6132 so that interfacing the Z80A CPU refresh control signals is not required.

The 74S74 flip-flop is useful for two reasons. The Z80A CPU refresh cycle, with its accompanying MREQ, is effectively blocked by the 74S74 during an M1 cycle. This is required because the refresh cycle during machine cycle M1 generates an MREQ signal that violates the AC timing requirements of the Z6132. The second purpose of the 74S74 is realized during an interrupt acknowledge cycle. The Z80A CPU uses the simultaneous occurrence of M1 active with IORQ active to indicate that an interrupt acknowledge cycle is in progress. If the 74S74 flip-flop is removed, the Address Clock becomes active during every clock cycle time (425 ns) for the Z6132-5. Figure 3 illustrates memory timing for

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>85</td>
<td>85</td>
<td>85</td>
</tr>
<tr>
<td>80</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>110</td>
<td>50+</td>
<td></td>
</tr>
<tr>
<td>150</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-15</td>
<td></td>
<td>READ DATA SETUP TIME</td>
</tr>
<tr>
<td>14</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>115</td>
<td>105</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3. Z80A Memory Cycle Timing
the Z80A CPU memory read or write cycle. In this cycle, MREQ is issued by the Z80A CPU to initiate a memory operation. The Z80A CPU control signals, MREQ and RD, closely track each other over the guaranteed temperature range. Were this not the case, DS could potentially become active before AC becomes true. The three 74LS04 inverters in the DS path help to insure that DS will become active only after AC has become true. Figure 3 shows WE in a memory read cycle. Only the occurrence of M1 (indicating an opcode fetch or an interrupt acknowledge) or the occurrence of RD (indicating M1 or memory read) inhibit WE from becoming active. During a memory read, the close tracking of MREQ and RD insures that WE setup time to AC High (—10 ns) is met.

Figure 4 shows a Z80A CPU I/O cycle along with the corresponding active Z6132 memory control signals. Since AC never makes a positive transition during this I/O cycle, the other memory control signals (such as CS and DS) do not affect operation of the Z6132. Figure 5 shows a Z80A CPU interrupt acknowledge cycle. Although AC makes a positive transition and CS could be true (depending on the Z80A CPU's current PC), the memory control signal DS never becomes active during an interrupt acknowledge cycle. This cycle appears to be an aborted read cycle to the Z6132 and has no harmful effect.

Thus, with only three commonly available 14-pin packages, a simple interface between the Z80A CPU and the Z6132 can be constructed. The Z80A was chosen for this application example because it allows 4 MHz operation while using relatively inexpensive (300 ns) memory. Operation of the Z80B CPU (6 MHz) provides for a maximum memory access time of 210 ns in the opcode fetch cycle (not including memory interface logic) under worst-case conditions. Figure 6 shows the timing for the Z80B opcode fetch cycle with its associated

Figure 4. Z80A I/O Cycle Timing
maximum delays. In this configuration, one wait state can be inserted to increase the available access time to 375 ns. In systems that require higher performance, the Z80B CPU (even with one wait state included in opcode fetch cycles) can increase processor execution efficiency. The Z80 CPU (2.5 MHz) is also easily interfaced with the Z6132 family. Here, as with the Z80A CPU, no additional wait states need to be added.

**Figure 5. Z80A Interrupt Acknowledge Cycle Timing**

**Figure 6. Z80B Opcode Fetch Timing**
Two Z6132s are interfaced to a Z8002 (non-segmented Z8000) in this example to provide 4K words (16 bits wide) of buffer storage. Three external TTL packages provide all address chip select and byte/word decoding to the Z6132s. The timing diagrams (Figures 8-10), the interface logic (Figure 11), and the circuit description are discussed later. Information on the Z8002 CPU can be obtained from the Z8000 CPU Product Specification (document number 00-2045-A), the Z8001/Z8002 CPU AC Characteristics (document number 00-2004-A) and from the Z8000 CPU Technical Manual (document number 00-2010-C). A Z8002 running at 4 MHz was chosen to provide high throughput while still providing a generous memory access time of 360 ns for the Z6132s (Figures 8-10). The Z6132-6 chosen for this example has a maximum access time of 350 ns. All Z8002 memory transactions are three clock cycles long and conform to the Zilog Z-BUS timing specifications. More information on the Zilog Z-BUS can be found in the Z-BUS Summary (document number 00-2031-A).
Interfacing the Z6132 to the Z8002 CPU (Continued)

The Z8002 uses a multiplexed address/data bus to provide for memory addressing and data transfer. The rising edge of Address Strobe (AS) guarantees that addresses from the Z8002 are stable. This signal (AS) is fed directly to the Z6132 as the Address Clock (AC) input clocks in memory addresses and initiates a memory cycle. The Z6132 samples its Chip Select (CS) pin with the rising edge of AC to determine whether the bus transaction is intended for it. If CS is found Low on the rising edge of AC, the Z6132 begins a read or write operation, depending on the state of its Write Enable (WE) pin. The Z6132 samples WE again on the falling edge of Data Strobe (DS). If WE is still Low, the write cycle is continued. If WE has returned to the High state, the memory write cycle to the Z6132 is aborted. This feature of the Z6132 allows memory write cycles to be suppressed if determined undesirable, without paying an access-time penalty. The R/W signal is fed directly from the Z8002 to the Z6132 WE pin. The signal DS from the Z8002 indicates when valid data is available on the multiplexed adress/data bus. This signal indicates if valid CPU data is available to the Z6132 during a write cycle and enables the Z6132 output buffers during a CPU read cycle. The DS signal from the CPU is fed directly to the DS input of the Z6132. The only interface circuitry between the Z8002 and the Z6132 is the decoding of required byte/word, read/write, and high-byte/low-byte Z8002 memory control functions (Figure 11). A 74LS157 dual multiplexer is used to provide enable signals for the even and odd banks of Z6132s. The truth table for this multiplexer follows. Both even and odd banks are enabled except during byte operations. During byte write operations, only one bank of Z6132s is enabled. This bank is determined by AD0.

<table>
<thead>
<tr>
<th>INPUTS</th>
<th>OUTPUTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/W</td>
<td>AD0</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

X = don't care

When the Z8002 performs a read operation, 16 bits of memory data are returned to the CPU. For byte read transactions, the appropriate (odd or even) byte is selected internally to the Z8002. The enable input for the 74LS157 is active Low. When the R/W output of the Z8002 is High (indicating a read operation), the 74LS157 is disabled, forcing the even and odd outputs Low. During a write operation, the 74LS157 is enabled and the even and odd outputs are determined by the states of the B/W and AD0 CPU outputs. During a word-write operation, both even and odd outputs are enabled. During a byte-write operation, the enabled even or odd bank is determined by the least significant address bit (AD0). A byte-write to an even address (AD0 is 0) corresponds to an even enable. When this byte is
read back from the Z6132, the Z8002 expects it to appear on the upper eight data bits and the Z8002 CPU (AD15-AD8). The lower eight data bits are connected to the odd bank, and the upper eight data bits are connected to the even bank. The least significant address bit (AD0) is not connected to the Z6132 (although it still functions as a data bit). It is used instead in the selection of even or odd Z6132 banks. A 74LS138 is used to further decode even and odd addresses into individual even and odd Chip Selects for the Z6132s. Memory transactions (excluding refresh operations) are reflected by status bit 03 (High) of the Z8002 CPU. This bit is fed to

Figure 9. Z8000 Memory Transaction (4.0 MHz)

Figure 10. Z6132-6 Interface Timing (4.0 MHz)
Interfacing the 74LS138s as a Chip Enable to inhibit memory Chip Selects during I/O operations that might correspond to the same address as one of the Z6132s.

The only timing parameter that requires explanation in this interface is the Chip Select timing (Figure 10) pertaining to the Z6132 Address Clock. All other timing parameters shown are generated by the Z8002 (no buffering is included) and meet the required setup, delay, and hold times for the Z6132. The Z8000 guarantees at least 55 ns delay from memory addresses stable to the rising edge of Address strobe (AS from the Z8002, AC to the Z6132). The worst-case timing condition for Chip Select to the Z6132 occurs during byte-write memory transactions. A maximum delay of 21 ns is introduced in the 74LS157 from the R/W input to the odd or even outputs. The 74LS138 decoders add a maximum worst-case delay of 32 ns from the decoder enable to the decoder outputs. The total byte-write Chip Select delay from address stable (21 ns + 32 ns) comes to 53 ns under worst-case timing considerations. Since the Chip Select setup time to AC is 0 ns, the CS-to-AC requirements for the Z6132 are satisfied.

Thus, with three low-power Schottky TTL packages, the Z8002 can access up to 64K bytes of primary memory in 8K-byte increments. The lowest 4K bytes are usually reserved for bootstrap ROM, but circuitry could be included to disable the ROM after bootstrap to provide a full 64K bytes of Z6132 RAM storage. The memory transaction timing diagram for 6-MHz Z8002 operation is included in Figure 8 for high-performance Z8002 designs. The Z6132-3 has a guaranteed access time of 200 ns and is suggested for use with the 6-MHz Z8001 or Z8002.

Figure 11. Z8000/Z6132 Interface Logic
In the following example, a Z6132-5 (300 ns) is interfaced to a Z8 operating at 7.3728 MHz. Timing for interfacing the Z8 to a Z6132-4 (250 ns) is discussed for 8-MHz Z8 operation. In addition, the example describes 2716 and 2732 EPROM interchangeability with the Z6132. Timing diagrams and circuit drawings have been included for Z8 memory interface timing and are discussed in this section.

The Z8 is an 8-bit, general-purpose microcomputer chip that can be configured under software control. The Z8 features regular architecture with 144 on-chip registers, 2K bytes of on-chip ROM, and 32 I/O lines configured for conventional I/O or for external memory. Detailed information on the Z8 can be found in the Z8 Microcomputer Technical Manual (document number 03-3047-02) and the Z8601/2/3 MCU Microcomputer Product Specification (document number 00-2037-A). The Z8 uses Port I (eight bits wide) as a multiplexed address/data bus and Port 0 as the upper byte of a 16-bit address bus. Before external memory references to the Z6132 can be made by any instruction, the user must configure Ports 0 and 1 appropriately. Instruction pipelining mandates that after setting the modes of Ports 0 and 1 for external memory operation, the next two bytes are fetched from internal program memory. Two single-byte instructions, such as NOPs, can be used to accomplish this. On-board ROM in the Z8 is available from 0000-07FF (Hex). This application locates the external Z6132 in the Z8 address space from 1000-1FFF (Hex).

All Z8 timing references are made with respect to the output signals AS and DS. The control signal AS indicates when the Z8 address bus is valid, while the control signal DS controls the flow of data. The Z8 status signal R/W (Read/Write) indicates the direction of data flow. The Z8 indicates when a

Figure 12. Z8601/Z6132 Interface Logic
Interfacing the Z6132 to the Z8 (Continued)

Hardware reset operation is in progress by activating DS while outputting AS at the internal clock rate. Since the internal clock has a cycle time period of 250 ns, it is necessary to inhibit AS during hardware reset operations so that the minimum memory cycle time for the Z6132 is not violated. This is easily accomplished by using the reset line to the Z8 as an inhibit line to the AC input of the Z6132 (Figure 12). The 74LS32 OR gate delays the Address Clock to the Z6132 a maximum of 22 ns.

EPROM Compatibility

The Z6132 is packaged in an industry-standard, 28-pin DIP and is pin compatible with the proposed JEDEC standard. This allows the substitution of other 28-pin DIPs that conform to the proposed JEDEC standard (namely the 2732 and 2716 EPROMs). The 2732 EPROM requires only that +5 V (VCC) be substituted for AC (pin 26 on the Z6132). This substitution can be accomplished easily with a jumper (Figure 12). Interfacing a 2716 requires one additional jumper change. The 2716 EPROM is only 2K bytes, and hence requires only 11 address bits for full addressing capability. A second jumper pad for 2716 selection can be included to tie pin 23 to a pullup resistor as required for reading a 2716.

Since the Z8 multiplexes addresses and data on Port 1, it is necessary to latch the low-order address byte with the Z8 control signal AS. This latch is unnecessary for systems without 2716/2732 EPROM capability, since the address to the Z6132 may change after the specified address hold time (60 ns for the Z6132-5). The 2716 and 2732 EPROMs are 24-pin packages, and the Z6132 is a 28-pin package. This requires the EPROMs to be physically justified so that pin 1 of the 2716/2732 is aligned with pin 3 of the Z6132.

Theory of Operation

Figure 12 shows the circuit diagram for a small Z8 system. In this configuration, a series resonant crystal (7.3728 MHz) provides all system timing. Port 1 is configured for multiplexed address and data, and Port 0 is configured to provide the upper address byte to complete the 12-bit address bus required by the Z6132 and to provide four bits of address decoding. The upper bits of Port 0 (P04 to P07) are decoded by a 74S138 to provide eight blocks, each 4K bytes long. The first block is discarded because it overlaps with internal 28 ROM. The second segment is used to generate CS for the Z6132, and the last six segments are free for other system chip select decoding, such as additional memory or external I/O ports. A 74LS373 is used to latch addresses from the multiplexed address/data bus of Port 1. This latch is enabled when AS is active (Low) and retains the addresses after AS has returned High. The Z6132 does not require addresses to be stable throughout the entire memory cycle, so this latch is used only with systems that provide the option of using the 2716 and 2732 EPROMs. Addresses are latched internally to the Z6132 on the rising edge of AC. Jumpers J1 and J2 are connected as shown for Z6132 operation. To substitute a 2732 for the Z6132, the existing jumper (J1) must be cut from the Z6132 pin 26 to the Z8 pin 9, and Z6132 pin 26 is connected to VCC. To substitute a 2716, one additional jumper change must be made. Jumper J2 is shown connected for Z6132 and 2732 operation. To substitute a 2716, the existing jumper is cut from the Z6132 pin 23 to the Z8 pin 16, and the jumper at J2 from the Z8 pin 23 is connected to the 4.7K pullup resistor.
The important control signals for memory interface to the Z8 have been reproduced in Figures 13-16. In this design example, a crystal frequency of 7.3728 MHz was selected for overall system timing. The Z8 product specifications provide timing specifications at 8 MHz. To calculate the timing parameters for frequencies other than 8 MHz, the timing parameters are derated by a factor based on the difference in clock period. For instance, the timing parameter TdA(AS) is given as 30 ns (min) for a clock input of 8 MHz. To calculate the timing value for a clock input of 7.3728 MHz, the difference in clock periods (135.6 ns - 125.0 ns = 10.6 ns) must be added to the value given in the Z8 product specifications. Hence, the delay time for TdA(AS) with a 7.3728 MHz clock is 40.6 ns (30 ns + 10.6 ns = 40.6 ns). The AS signal has a guaranteed minimum width of 70.6 ns at 7.3728 MHz. The Z8 guarantees that addresses will be stable 40.6 ns before the rising edge of AS. With the additional maximum delay of 22 ns for the 74LS32, the resultant delay (AS)
Timing (Continued) is fed directly to the Address Clock input of the Z6132. The low-byte address encounters a maximum delay of 30 ns through the 74LS373 latch. The status signal R/W and the data bus control signal DS are fed directly to the Z6132. The status signal R/W is available to the Z6132 40.6 ns before the rising edge of AC. The maximum delay for CS through the 74S138 is 12 ns. This still leaves 27.4 ns setup time for CS to AC, although 0 ns is the minimum requirement. The maximum access time for an external memory operation at 7.3728 MHz is calculated to be 322.4 ns (Figure 14). This access time begins with the rising edge of AC and includes the data setup time to the Z8 CPU. This access time allows the use of low-speed Z6132-5 (300 ns) RAMs. For systems that require higher performance, the Z6132-4 can be used with an 8-MHz Z8 CPU. Timing for the Z8 at 8 MHz has been included in Figure 13. The maximum access time allowed for external RAM by the Z8 when operating at 8 MHz is 280 ns. The Z6132-4 has an access time of 250 ns, making it directly compatible with an 8-MHz Z8.

![Figure 15. Z8132 Memory Timing (7.3728 MHz)](image)

![Figure 16. External Memory Timing (7.3728 MHz)](image)
A Minimum Z8 System

Figure 17 illustrates the simplicity with which a Z8601/Z6132 system is reduced to a minimum chip count. The expansion bus of the Z8601 and the interface to the Z6132 are Z-BUS compatible. As a result, the two parts connect directly without additional logic. As mentioned in the previous section, the access time of the Z6132-4 meets the requirements of an 8-MHz Z8.

Summary

The Z6132 is a versatile, intelligent byte-wide RAM, which provides an attractive solution for primary buffer storage. Because the Z6132 provides two modes of self-refresh, the user can select between executing a refresh after each memory access or taking advantage of the inherent sequential access of most memory systems. The Z6132 is an industry-standard, 28-pin DIP that conforms to the JEDEC recommended pinout and is interchangeable with 2716/2732-type EPROMs. The Z6132 is Z-BUS compatible and interfaces easily with the Z8, Z80, and Z8000 Families of microprocessors.

Figure 17. Z8601/Z6132 Minimum System
<table>
<thead>
<tr>
<th>Z-Bus 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zilog</td>
</tr>
<tr>
<td>Zilog</td>
</tr>
<tr>
<td>Zilog</td>
</tr>
<tr>
<td>Zilog</td>
</tr>
</tbody>
</table>
Z-BUS™
Component Interconnect

Summary

March 1981

Features
- Multiplexed address/data bus shared by memory and I/O transfers.
- 16 or more memory address bits; 16-bit I/O addresses; 8 or 16 data bits.
- Supports polling and vectored or non-vectored interrupts.
- Daisy-chain interrupt structure services interrupts without a separate priority controller.

General Description

The Z-BUS is a high-speed parallel shared bus that links components of the Z8000 Family. It provides family members with a common communication interface that supports the following kinds of interactions:

- **Data Transfer.** Data can be moved between bus controllers (such as a CPU) and memories or peripherals.
- **Interrupts.** Interrupts can be generated by peripherals and serviced by CPUs over the bus.
- **Resource Control.** Distributed management of shared resources (including the bus itself) is supported by a daisy-chain priority mechanism.

The heart of the Z-BUS is a set of multiplexed address/data lines and the signals that control these lines. Multiplexing data and addresses onto the same lines makes more efficient use of pins and facilitates expansion of the number of data and address bits. Multiplexing also allows straightforward addressing of a peripheral's internal registers, which greatly simplifies I/O programming.

A daisy-chained priority mechanism resolves interrupt and resource requests, thus allowing distributed control of the bus and eliminating the need for separate priority controllers. The resource-control daisy chain allows wide physical separation of components.

The Z-BUS is asynchronous in the sense that peripherals do not need to be synchronized with the CPU clock. All timing information is provided by Z-BUS signals.

---

**Figure 1. Z-BUS Signals**
Z-BUS Components

A Z-BUS component is one that uses Z-BUS signals and protocols, and meets the specified ac and dc characteristics. Most components in the Z8000 Family are Z-BUS components. The four categories of Z-BUS components are as follows:

- **CPUs.** A Z-BUS system contains one CPU, and this CPU has default control of the bus and typically initiates most bus transactions. Besides generating bus transactions, it handles interrupt and bus-control requests. The Z8001 Segmented CPU and Z8002 Non-Segmented CPU are Z-BUS CPUs.

- **Peripherals.** A Z-BUS peripheral is a component capable of responding to I/O transactions and generating interrupt requests. The Z8036 Counter Input/Output Circuit (Z-CIO), Z8038 FIFO Input/Output, Interface Unit (Z-FIO), the Z8030 Serial Communication Controller (Z-SCC), the Z8090 Universal Peripheral Controller (Z-UPC), and the Z8052 CRT Controller (Z-CRT) are all Z-BUS peripherals.

- **Requesters.** A Z-BUS requester is any component capable of requesting control of the bus and initiating transactions on the bus. A Z-BUS requester is usually also a peripheral. The Z8016 DMA Transfer Controller (Z-DTC) is a Z-BUS requester and a peripheral.

- **Memories.** A Z-BUS memory is one that interfaces directly to the Z-BUS and is capable of fetching and storing data in response to Z-BUS memory transactions. The Z6132 Quasi-Static RAM is a Z-BUS memory.

Other Components

The Z8 Microcomputer—in its microprocessor configuration—conforms to Z-BUS timing (which allows it to use Z-BUS peripherals and memories), but is missing a wait input and certain status outputs.

The Z8010 Memory Management Unit (Z-MMU) is a Z8000 CPU support component that interfaces with part of the Z-BUS on the CPU side and provides demultiplexed addresses on the memory side.

The Z8060 First-In-First-Out Buffer (Z-FIFO) is not a Z-BUS component; rather, it is used to expand the buffer depth of the Z-FIO or to interface the I/O ports of the Z-UPC, Z-CIO, or Z-FIO to user equipment.

Z-80 Family components, while not Z-BUS compatible, are easily interfaced to Z-BUS CPUs.

Operation

Two kinds of operations can occur on the Z-BUS: transactions and requests. At any given time, one device (either the CPU or a bus requester) has control of the Z-BUS and is known as the bus master. A transaction is initiated by a bus master and is responded to by some other device on the bus. Four kinds of transactions occur in Z-BUS systems:

- **Memory.** Transfers 8 or 16 bits of data to or from a memory location.
- **I/O.** Transfers 8 or 16 bits of data to or from a peripheral.
- **Interrupt Acknowledge.** Acknowledges an interrupt and transfers an identification/status vector from the interrupting peripheral.
- **Null.** Does not transfer data. Typically used for refreshing memory.

Only one transaction can proceed on the bus at a time, and it must be initiated by the bus master. A request, however, may be initiated by a component that does not have control of the bus. There are three kinds of requests:

- **Interrupt.** Requests the attention of the Z-BUS CPU.
- **Bus.** Requests control of the Z-BUS to initiate transactions.
- **Resource.** Requests control of a particular resource.

When a request is made, it is answered according to its type: for interrupt requests an interrupt-acknowledge transaction is initiated; for bus and resource requests an acknowledge signal is sent. In all cases a daisy-chain priority mechanism provides arbitration between simultaneous requests.
The Z-BUS consists of a set of common signal lines that interconnect bus components (Figure 1). The signals on these lines can be grouped into four categories, depending on how they are used in transactions and requests.

**Primary Signals.** These signals provide timing, control, and data transfer for Z-BUS transactions.

**AD0–AD15. Address/Data (active High).** These multiplexed data and address lines carry I/O addresses, memory addresses, and data during Z-BUS transactions. A Z-BUS may have 8 or 16 bits of data depending on the type of CPU. In the case of an 8-bit Z-BUS, data is transferred on AD0–AD7.

**Extended Address. (active High).** These lines extend AD0–AD15 to support memory addresses greater than 16 bits. The number of lines and the type of address information carried is dependent on the CPU.

**Status. (active High).** These lines designate the kind of transaction occurring on the bus and certain additional information about the transaction (such as program or data memory access or System versus Normal Mode).

**AS. Address Strobe (active Low).** The rising edge of AS indicates the beginning of a transaction and that the Address, Status, R/W, and B/W signals are valid.

**DS. Data Strobe (active Low).** DS provides timing for data movement to or from the bus master.

**R/W. Read/Write (Low = write).** This signal determines the direction of data transfer for memory or I/O transactions.

**B/W. Byte/Word (Low = word).** This signal indicates whether a byte or word of data is to be transmitted on a 16-bit bus. This signal is not present on an 8-bit bus.

**WAIT. (active Low).** A Low on this line indicates that the responding device needs more time to complete a transaction.

**RESET. (active Low).** A Low on this line resets the CPU and bus users. Peripherals may be reset by RESET or by holding AS and DS Low simultaneously.

**CS. Chip Select (active Low).** Each peripheral or memory component has a CS line that is decoded from the address and status lines. A Low on this line indicates that the peripheral or memory component is being addressed by a transaction. The Chip Select information is latched on the rising edge of AS.

**CLOCK.** This signal provides basic timing for bus transactions. Bus masters must provide all signals synchronously to the clock. Peripherals and memories do not need to be synchronized to the clock.

**Bus Request Signals.** These signals make bus requests and establish which component should obtain control of the bus.

**BUSRQ. Bus Request (active Low).** This line is driven by all bus requesters. A Low indicates that a bus requester has or is trying to obtain control of the bus.

**BUSACK. Bus Acknowledge (active Low).** A Low on this line indicates that the Z-BUS CPU has relinquished control of the bus in response to a bus request.

**BAI, BAO. Bus Acknowledge In, Bus Acknowledge Out (active Low).** These signals form the bus-request daisy chain.
<table>
<thead>
<tr>
<th>Z-BUS Connections</th>
<th>Signal</th>
<th>CPU</th>
<th>Requester</th>
<th>Peripheral</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z-BUS Connections</td>
<td>AD0-AD15</td>
<td>Bidirectional(^2)</td>
<td>Bidirectional(^2)</td>
<td>Bidirectional(^1)</td>
<td>Bidirectional(^2)</td>
</tr>
<tr>
<td></td>
<td>3-state</td>
<td>3-state</td>
<td>3-state</td>
<td>3-state</td>
<td></td>
</tr>
<tr>
<td>Extended Address(^8)</td>
<td>Output</td>
<td>Output</td>
<td>Output</td>
<td>Input</td>
<td></td>
</tr>
<tr>
<td>Status</td>
<td>Output</td>
<td>Output</td>
<td>Output</td>
<td>Input(^10)</td>
<td></td>
</tr>
<tr>
<td>R/W</td>
<td>Output</td>
<td>Output</td>
<td>Output</td>
<td>Input</td>
<td></td>
</tr>
<tr>
<td>B/W(^9)</td>
<td>Output</td>
<td>Output</td>
<td>Input(^3)</td>
<td>Input</td>
<td></td>
</tr>
<tr>
<td>WAIT</td>
<td>Input</td>
<td>Input</td>
<td>Output(^6)</td>
<td>Output(^8)</td>
<td></td>
</tr>
<tr>
<td>AS</td>
<td>Output</td>
<td>Output</td>
<td>Input</td>
<td>Input</td>
<td></td>
</tr>
<tr>
<td>DS</td>
<td>Output</td>
<td>Output</td>
<td>Input</td>
<td>Input</td>
<td></td>
</tr>
<tr>
<td>CS(^4)</td>
<td>Input</td>
<td>Input</td>
<td>Input</td>
<td>Input</td>
<td></td>
</tr>
<tr>
<td>RESET</td>
<td>Input</td>
<td>Input(^13)</td>
<td>Input(^8)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLOCK(^14)</td>
<td>Input</td>
<td>Input</td>
<td>Input(^8)</td>
<td>Input(^8)</td>
<td></td>
</tr>
<tr>
<td>BUSREQ</td>
<td>Input</td>
<td>Bidirectional</td>
<td>Open Drain</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BUSACK</td>
<td>Output</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BA(^7)</td>
<td>Input</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BAO(^7)</td>
<td>Output</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INT</td>
<td>Input</td>
<td></td>
<td>Output</td>
<td>Open Drain</td>
<td></td>
</tr>
<tr>
<td>INTACK(^6)</td>
<td></td>
<td></td>
<td>Input(^11)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IE(^7)</td>
<td></td>
<td></td>
<td>Input</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IEO(^7)</td>
<td></td>
<td></td>
<td>Output</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMRQ(^12)</td>
<td>Output</td>
<td>Open Drain</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMST(^12)</td>
<td>Input</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MMAI(^7, 12)</td>
<td>Input</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MМАО(^7, 12)</td>
<td>Output</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Only AD0-AD7, unless peripheral is 16-Bit.
2. For an 8-bit bus, only AD0-AD7 are bidirectional.
3. Only for a 16-bit peripheral.
4. Derived signal, one for each peripheral or memory; decoded from status and address lines.
5. Optional—peripherals are typically reset by AS and DS being Low simultaneously; however, they can have a reset input.
6. Derived signal; decoded from status lines.
7. Daisy-chain lines.
8. Optional signal(s).
9. For 16-bit data bus only.
10. Optional—usually only input on peripherals that are also requesters.
11. May be omitted if peripheral inputs status lines.
12. Optional signal; any component may attach to the resource request lines.
13. Optional signal; a bus requestor may also be reset by AS and DS going Low and BA being High simultaneously.
14. This signal is optional if there are no requesters on the bus. CPU timing can be provided by alternate means such as crystal oscillator inputs.

Table 1. Z-BUS Component Connections to Signal Lines. This table shows how the various Z-BUS components attach to each signal line. When a device is both a bus requester and a peripheral, the attributes in both columns of the table should be combined (e.g., input combined with output and 3-state becomes bidirectional and 3-state.)
**Interrupt Signals.** These signals are used for interrupt requests and for determining which interrupting component is to respond to an acknowledge. To support more than one type of interrupt, the lines carrying these signals can be replicated. (The Z8000 CPU supports three types of interrupts: non-maskable, vectored, and non-vectored.)

**INT. Interrupt (active Low).** This signal can be driven by any peripheral capable of generating an interrupt. A Low on INT indicates that an interrupt request is being made.

**INTACK. Interrupt Acknowledge (active Low).** This signal is decoded from the status lines. A Low indicates an interrupt acknowledge transaction is in progress. This signal is latched by the peripheral on the rising edge of AS.

**IEI, IEO. Interrupt Enable In, Interrupt Enable Out (active High).** These signals form the interrupt daisy chain.

**Resource Request Signals.** These signals are used for resource requests. To manage more than one resource, the lines carrying these signals can be replicated. (The Z8000 supports one set of resource request lines.)

**MMRQ. Multi-Micro Request (active Low).** This line is driven by any device that can use the shared resource. A Low indicates that a request for the resource has been made or granted.

**MMST. Multi-Micro Status (active Low).** This pin allows a device to observe the value of the MMRQ line. An input pin other than MMRQ facilitates the use of line drivers for MMRQ.

**MMAI, MMAO. Multi-Micro Acknowledge In, Multi-Micro Acknowledge Out (active Low).** These lines form the resource-request daisy chain.

---

**Transactions**

All transactions start with Address Strobe being driven Low and then raised High by the bus master (Figure 2). The Status lines are valid on the rising edge of Address Strobe and indicate the type of transactions being initiated. If the transaction requires an address, it must also be valid on the rising edge of Address Strobe.

For all transactions except null transactions (which do nothing beyond this point), data is then transferred to or from the bus master. The bus master uses Data Strobe to time the movement of data. For a read (R/W = High), the bus master makes AD0-AD15 inactive before driving Data Strobe Low so that the addressed memory or peripheral can put its data on the bus. The bus master samples this data just before raising Data Strobe High. For a write (R/W = Low), the bus master puts the data to be written on AD0-AD15 before forcing Data Strobe Low.

For an 8-bit 2-BUS, data is transferred on AD0-AD7. Address bits may remain on AD8-AD15 while DS is Low.

---

![Figure 2. Typical Transaction Timing](image-url)
Memory Transactions

For a memory transaction, the Status lines distinguish among various address spaces, such as program and data or system and normal, as well as indicating the type of transaction. The memory address is put on AD0–AD15 and on the extended address lines.

For a Z-BUS with 16-bit data, the memory is organized as two banks of eight bits each (Figure 3). One bank contains all the upper bytes of all the addressable 16-bit words. The other bank contains all the lower bytes. When a single byte is written (R/W = Low, B/W = High), only the bank indicated by address bit A0 is enabled for writing.

For a Z-BUS with 8-bit data, the memory is organized as one bank which contains all bytes. This bank always inputs and outputs its data on AD0–AD7.

I/O transactions are similar to memory transactions with two important differences. The first is that I/O transactions take an extra clock cycle to allow for slow peripheral operation. The second is that byte data (indicated by B/W High on a 16-bit bus) is always transmitted on AD0–AD7, regardless of the I/O address. (AD8–AD15 contain arbitrary data in this case.) For an I/O transaction, the address indicates a peripheral and a particular register or function within that peripheral.

Null Transactions

The two kinds of null transactions are distinguished by the Status lines: internal operation and memory refresh. Both transactions look like a memory read transaction except that Data Strobe remains High and no data is transferred.

For an internal operation transaction, the Address lines contain arbitrary data when Address Strobe goes High. This transaction is inititated to maintain a minimum transaction rate when a bus master is doing a long internal operation (to support memories which generate refresh cycles from Address Strobe).

For a memory refresh transaction, the Address lines contain a refresh address when Address Strobe goes High. This transaction is used to refresh a row of a dynamic memory.

Any memory or I/O transaction can be suppressed (effectively turning it into a null transaction) by keeping Data Strobe High throughout the transaction.

Interrupts

A complete interrupt cycle consists of an interrupt request followed by an interrupt-acknowledge transaction. The request, which consists of INT pulled Low by a peripheral, notifies the CPU that an interrupt is pending. The interrupt-acknowledge transaction, which is initiated by the CPU as a result of the request, performs two functions: it selects the peripheral whose interrupt is to be acknowledged, and it obtains a vector that identifies the selected device and cause of interrupt.

A peripheral can have one or more sources of interrupt. Each interrupt source has three bits that control how it generates interrupts. These bits are an Interrupt Pending bit (IP), and Interrupt Enable bit (IE), and an Interrupt Under Service bit (IUS). A peripheral may also have one or more vectors for identifying the source of an interrupt during an interrupt-acknowledge transaction. Each interrupt source is associated with one interrupt vector and each interrupt vector can have one or more interrupt sources associated with it. Each vector has a Vector Includes Status bit (VIS) controlling its use.

Finally, each peripheral has three bits for...
interrupts
(Continued)
controlling interrupt behavior for the whole
device. These are a Master Interrupt Enable
bit (MIE), a Disable Lower Chain bit (DLC),
and a No Vector bit (NV).

Peripherals are connected together via an
interrupt daisy chain formed with their IEI and
IEO pins (Figure 4). The interrupt sources
within a device are similarly connected into
this chain with the overall effect being a daisy
chain connecting the interrupt sources. The
daisy chain has two functions: during an
interrupt-acknowledge transaction, it deter­
mines which interrupt source is being
acknowledged; at all other times it determines
which interrupt sources can initiate an inter­
rupt request.

Figure 5 is a state diagram for interrupt
processing for an interrupt source (assuming
its IE bit is 1). An interrupt source with an
interrupt pending (IP = 1) makes an interrupt
request (by pulling INT Low) if, and only if, it
is enabled (IE = 1, MIE = 1), it does not have
an interrupt under service (IUS = 0), no
higher priority interrupt is being serviced
(IEI = High), and no interrupt-acknowledge
transaction is in progress (as indicated by
INTACK at the last rising edge of AS). IEO is
not pulled down by the interrupt source at this
time; IEO continues to follow IEI until an
interrupt-acknowledge transaction occurs.

Some time after INT has been pulled Low,
the CPU initiates an interrupt-acknowledge
transaction (indicated by INTACK Low).
Between the rising edge of AS and the falling
edge of DS, the IEI/IEO daisy chain settles.
Any interrupt source with an interrupt pending
(IP = 1, IE = 1, MIE = 1) or under service
(IUS = 1) holds its IEO line Low; all other
interrupt sources make IEO follow IEI. When
DS falls, only the highest priority interrupt
source with a pending interrupt (IP = 1) has
its IEI input High, its IE bit set to 1, and its
IUS bit set to 0. This is the interrupt source
being acknowledged, and at this point it sets
its IUS bit to 1, and, if the peripheral's NV bit
is 0, identifies itself by placing the vector on
AD0-AD7. If the NV bit is 1, then the periph­
eral’s AD0-AD7 pins remain floating, thus
allowing external circuitry to supply the vec­
tor. (All interrupts, including the Z8000's
non-vectored interrupt, need a vector for identify­
ing the source of an interrupt.) If the vector's
VIS bit is 1, the vector will also contain status
information further identifying the source of
the interrupt. If the VIS bit is 0, the vector
held in the peripheral will be output without
modification.

While an interrupt source has an interrupt
under service (IUS = 1), it prevents all lower
priority interrupt sources from requesting
interrupts by forcing IEO Low. When interrupt
servicing is complete, the CPU must reset the
IUS bit and, in most cases, the IP bit (by
means of an I/O transaction).

Figure 4. Interrupt Connections
Interrupts
(Continued)

Interrupts
(Continued)

Figure 5. State Diagram for an Interrupt Source

Transition Legend

A. The peripheral detects an interrupt condition and sets Interrupt Pending.
B. All higher priority peripherals finish interrupt service, thus allowing IEI to go High.
C. An interrupt-acknowledge transaction starts, and the EEI/IEO daisy chain settles.
D. The interrupt-acknowledge transaction terminates with the peripheral selected. Interrupt Under Service (IUS) is set to 1, and Interrupt Pending (IP) may or may not be reset.
E. The interrupt-acknowledge transaction terminates with a higher priority device having been selected.
F. The Interrupt Pending bit in the peripheral is reset by an I/O operation.
G. A new interrupt condition is detected by the peripheral, causing IP to be set again.
H. Interrupt service is terminated for the peripheral by resetting IUS.
I. IE is reset to zero, causing interrupts to be disabled.
J. IE is set to one, re-enabling interrupts.

State Legend

0. No interrupts are pending or under service for this peripheral.
1. An interrupt is pending, and an interrupt request has been made by pulling INT Low.
2. An interrupt is pending, but no interrupt request has been made because a higher priority peripheral has an interrupt under service, and this has forced IEI Low.
3. An interrupt-acknowledge sequence is in progress, and no higher priority peripheral has a pending interrupt.
4. An interrupt-acknowledge sequence is in progress, but a higher priority peripheral has a pending interrupt, forcing IEI Low.
5. The peripheral has an interrupt under service. Service may be temporarily suspended (indicated by IEI going Low) if a higher priority device generates an interrupt.
6. This is the same as State 5 except that an interrupt is also pending in the peripheral.
7. Interrupts are disabled from this source because IE = 0.
8. Interrupts are disabled from this source and lower priority sources because IE = 0 and IUS = 1.

1. This diagram assumes MIE = 1. The effect of MIE = 0 is the same as that of setting IE = 0.
2. The DLC bit does not affect the states of individual interrupt sources. Its only effect is on the IEO output of a whole peripheral.
3. Transition 1 to state 6 or 7 can occur from any state except 3 or 4 (which only occur during interrupt acknowledge).
4. Transition 1 from state 6 or 7 can be to any state except 3 or 4, depending on the value of IEI, IP, and IUS.
A peripheral’s Master Interrupt Enable bit (MIE) and Disable Lower Chain bit (DLC) can modify the behavior of the peripheral’s interrupt sources in the following way: if the MIE bit is 0, the effect is as if every Interrupt Enable bit (IE) in the peripheral were 0; thus all interrupts from the peripheral are disabled. If the DLC bit is 1, the effect is to force the peripheral’s IEO output Low, thus disabling all lower priority devices from initiating interrupt requests.

Polling can be done by disabling interrupts (using MIE and DLC) and by reading peripherals to detect pending interrupts. Each Z-BUS peripheral has a single directly addressable register that can be read to determine if there is an interrupt pending in the device and, if so, what interrupt source it is from.

To generate transactions on the bus, a bus requester must gain control of the bus by making a bus request. This is done by forcing BUSREQ Low (Figure 6). A bus request can be made only if BUSREQ is initially High (and has been for two clock cycles), indicating that the bus is controlled by the CPU and no other device is requesting it.

After BUSREQ is pulled Low, the Z-BUS CPU relinquishes the bus and indicates this condition by making BUSACK Low. The Low on BUSACK is propagated through the BAI/BAO daisy chain (Figure 6). BAI follows BAO for components not requesting the bus, and any component requesting the bus holds its BAO High, thereby locking out all lower priority users.

A bus requester gains control of the bus when its BAI input goes Low. When it is ready to relinquish the bus, it stops pulling BUSREQ Low and allows BAO to follow BAI. This permits lower priority devices that made simultaneous requests to gain control of the bus. When all simultaneously requesting devices have relinquished the bus, BUSREQ goes High, returning control of the bus to the CPU and allowing other devices to request it.

The protocol to be followed in making a bus request is shown in Figure 7.
Resource Requests

Resource requests are used to obtain control of a resource that is shared between several users. The resource can be a common bus, a common memory or any other resource. The requestor can be any component capable of implementing the request protocol.

Unlike the Z-BUS itself, no component has control of a general resource by default; every device must acquire the resource before using it. All devices sharing the general resource drive the MMRQ line (Figure 8). When Low, the MMRQ line indicates that the resource is being acquired or used by some device. The MMST pin allows each device to observe the state of the MMRQ line.

When MMRQ is High, a device may initiate a resource request by pulling MMRQ Low (Figure 9). The resulting Low on MMRQ is propagated through the MMAI/MMAO daisy chain. If a device is not requesting the resource, its MMAO output follows its MMAI input. Any device making a resource request forces its MMAO output High to deny use of the resource to lower priority devices.

A device gains control of the resource if its MMAI input is Low (and its MMAO output is High) after a sufficient delay to let the daisy chain settle. If the device does not obtain the resource after this short delay, it must stop pulling MMRQ Low and make another request at some later time when MMRQ is again High. When a device that has gained control of a resource is finished, it releases the resource by allowing MMRQ to go High.

The four unidirectional lines of the resource request chain allow the use of line drivers, thus facilitating connection of components separated by some distance. In the case of the Z8000 CPU, the four resource request lines may be mapped into the CPU MI and MO pins using the logic shown in Figure 10. With this configuration, the Multi-Micro Request Instruction (MREQ) performs a resource request.

![Figure 8. Resource Request Connections](image8.png)

![Figure 9. Resource Request Protocol](image9.png)

1. For any resource requested, this wait time must be less than the minimum wait time plus resource usage time of all other requesters.

![Figure 10. Bus Request Logic for Z8000](image10.png)
The timing characteristics given in this document reference 2.0 V as High and 0.8 V as Low. The following test load circuit is assumed. The effect of larger capacitive loadings can be calculated by delaying output signal transitions by 10 ns for each additional 50 pF of load up to a maximum 200 pF.

The following table states the dc characteristics for the input and output pins of Z-BUS components. All voltages are relative to ground.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter</th>
<th>Min</th>
<th>Max</th>
<th>Unit</th>
<th>Test Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_{IL}$</td>
<td>Input Low Voltage</td>
<td>-0.3</td>
<td>0.8</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>$V_{IH}$</td>
<td>Input High Voltage</td>
<td>2.0 $V_{CC}$</td>
<td>$V_{CC} + 0.3$</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>$V_{IHRESET}$</td>
<td>Input High Voltage on RESET pin</td>
<td>2.4 $V_{CC}$</td>
<td>$V_{CC} to 0.3$</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>$V_{OL}$</td>
<td>Output Low Voltage</td>
<td>0.4</td>
<td>2.4</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>$V_{OH}$</td>
<td>Output High Voltage</td>
<td>2.4</td>
<td>250$\mu$A</td>
<td>V</td>
<td></td>
</tr>
<tr>
<td>$I_{IL}$</td>
<td>Input Leakage Current</td>
<td>-10</td>
<td>+10</td>
<td>$\mu$A</td>
<td></td>
</tr>
<tr>
<td>$I_{OL}$</td>
<td>3-State Output Leakage Current in Float</td>
<td>-10</td>
<td>+10</td>
<td>$\mu$A</td>
<td></td>
</tr>
</tbody>
</table>

Capacitance

The following table gives maximum pin capacitance for Z-BUS components. Capacitance is specified at a frequency of 1 MHz over the temperature range of the component. Unused pins are returned to ground.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Parameter</th>
<th>Max (pF)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_{IN}$</td>
<td>Input Capacitance</td>
<td>10</td>
</tr>
<tr>
<td>$C_{OUT}$</td>
<td>Output Capacitance</td>
<td>15</td>
</tr>
<tr>
<td>$C_{I/O}$</td>
<td>Bidirectional Capacitance</td>
<td>15</td>
</tr>
</tbody>
</table>

Timing Diagrams

The following diagrams and tables give the timing for each kind of transaction (except null transactions). Timings are given separately for bus masters and for peripherals and memories and are intended to give the minimum timing requirements which a Z-BUS component must meet. An individual component will have more detailed and sometimes more stringent timing specifications. The differences between bus master timing and peripheral and memory timing allow for buffer and decoding circuit delays and for signal skew. The timing given for memories is a constraint on bus-compatible memories (like the Z6132 Quasi-Static RAM) and is not intended to constrain memory sub-systems constructed from conventional components.

Besides these timings, there is a requirement that at least 128 transactions be initiated in any 2 ms period. This accommodates memories that generate refresh cycles from Address Strobe.
Parameters 1-25 are common to all transactions.
## Bus Master Timing Parameters

<table>
<thead>
<tr>
<th>Number</th>
<th>Symbol</th>
<th>Parameter</th>
<th>Min (ns)</th>
<th>Max (ns)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TpC</td>
<td>Clock Period</td>
<td>250</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>TwCh</td>
<td>Clock High Width</td>
<td>105</td>
<td>1895</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>TwCl</td>
<td>Clock Low Width</td>
<td>105</td>
<td>1895</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>TIC</td>
<td>Clock Fall Time</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>TrC</td>
<td>Clock Rise Time</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>TdC(S)</td>
<td>Clock * To Status Valid Delay</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>TdC(ASr)</td>
<td>Clock * To AS * Delay</td>
<td>90</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>TdC(ASI)</td>
<td>Clock * To AS * Delay</td>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>TdS(AS)</td>
<td>Status Valid To AS * Delay</td>
<td>50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>TwAS</td>
<td>AS Low Width</td>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>TdDS(S)</td>
<td>DS * To Status Not Valid Delay</td>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>TdAS(DS)</td>
<td>AS * To DS * Delay</td>
<td>70</td>
<td>2095</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>TsDR(C)</td>
<td>Read Data To Clock * Setup Time</td>
<td>50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>TdC(DS)</td>
<td>Clock * To DS * Delay</td>
<td>70</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>TdDS(AS)</td>
<td>DS * To AS * Delay</td>
<td>70</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>TdC(Az)</td>
<td>Clock * To Address Float Delay</td>
<td>65</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>TdC(A)</td>
<td>Clock * To Address Valid Delay</td>
<td>90</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>TdA(AS)</td>
<td>Address Valid To AS * Delay</td>
<td>50</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>TdAS(A)</td>
<td>AS * To Address Not Valid Delay</td>
<td>60</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>TwA</td>
<td>Address Valid Width</td>
<td>150</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>ThDR(DS)</td>
<td>Read Data To DS * Hold Time</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>TdDS(A)</td>
<td>DS * To Address Active Delay</td>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>TdDS(DW)</td>
<td>DS * To Write Data Not Valid Delay</td>
<td>80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>TsW(C)</td>
<td>WAIT To Clock * Setup Time</td>
<td>50</td>
<td>2,5</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>ThW(C)</td>
<td>WAIT To Clock * Hold Time</td>
<td>0</td>
<td>2,5</td>
<td></td>
</tr>
</tbody>
</table>

### All Transactions

<table>
<thead>
<tr>
<th>Number</th>
<th>Symbol</th>
<th>Parameter</th>
<th>Min (ns)</th>
<th>Max (ns)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>TdAS(W)</td>
<td>AS * To WAIT Required Valid</td>
<td>90</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>TdC(DSR)</td>
<td>Clock * To DS (Read) * Delay</td>
<td>120</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>TdDSR(DR)</td>
<td>DS (Read) * To Read Data Required Valid</td>
<td>185</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>TwDSR</td>
<td>DS (Read) Low Width</td>
<td>250</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>TdAz(DSR)</td>
<td>Address Float to DS (Read) * Delay</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>TdAS(DR)</td>
<td>AS * To Read Data Required Valid</td>
<td>320</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>TdA(DR)</td>
<td>Address Valid To Read Data Required Valid</td>
<td>400</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>TdC(DSW)</td>
<td>Clock * To DS (Write) * Delay</td>
<td>95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>TwDSW</td>
<td>DS (Write) Low Width</td>
<td>160</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>TdDW(DSWr)</td>
<td>Write Data To DS (Write) * Delay</td>
<td>50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>TdDW(DSWr)</td>
<td>Write Data Valid To DS (Write) * Delay</td>
<td>230</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Memory Transactions

<table>
<thead>
<tr>
<th>Number</th>
<th>Symbol</th>
<th>Parameter</th>
<th>Min (ns)</th>
<th>Max (ns)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
<td>TdAS(DR)</td>
<td>AS * To Read Data Required Valid</td>
<td>570</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>TdA(DR)</td>
<td>Address Valid To Read Data Required Valid</td>
<td>650</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>TdAz(DSI)</td>
<td>Address Float To DS (I/O) *</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>TdC(DSI)</td>
<td>Clock * To DS (I/O) *</td>
<td>120</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>TdDSI(DR)</td>
<td>DS (I/O) * To Read Data Required Valid</td>
<td>320</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>TwDSI</td>
<td>DS (I/O) Low Width</td>
<td>400</td>
<td></td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>TdDW(DSI)</td>
<td>Write Data To DS (I/O) *</td>
<td>50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>TdDW(DSII)</td>
<td>Write Data To DS (I/O) * Delay</td>
<td>480</td>
<td></td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>TdAS(W)</td>
<td>AS * To WAIT Required Valid</td>
<td>340</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### I/O Transactions

<table>
<thead>
<tr>
<th>Number</th>
<th>Symbol</th>
<th>Parameter</th>
<th>Min (ns)</th>
<th>Max (ns)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>46</td>
<td>TdAS(DSA)</td>
<td>AS * To DS (Acknowledge) * Delay</td>
<td>960</td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>TdC(DSA)</td>
<td>Clock * To DS (Acknowledge) * Delay</td>
<td>120</td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>TdDSA(DR)</td>
<td>DS (Acknowledge) * To Read Data Required Valid</td>
<td>420</td>
<td></td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>TwDSA</td>
<td>DS (Acknowledge) Low Width</td>
<td>485</td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>TdAS(W)</td>
<td>AS * To Wait Required Valid</td>
<td>840</td>
<td></td>
<td></td>
</tr>
<tr>
<td>51</td>
<td>TdDSA(W)</td>
<td>DS (Acknowledge) * To Wait Required Valid</td>
<td>130</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Timing for extended addresses is CPU dependent; however, extended addresses must be valid at least as soon as addresses are valid on ADP_{0-15} and must remain valid at least as long as addresses are valid on ADP_{0-15}.
2. The exact clock cycle that wait is sampled on depends on the type of transaction; however, wait always has the given setup and hold times to the clock.
3. The maximum value for TdAS(DS) does not apply to Interrupt-Acknowledge Transactions.
4. Except where otherwise stated, maximum rise and fall times for inputs are 200 ns.
5. The setup and hold times for WAIT to the clock must be met. If WAIT is generated asynchronously to the clock, it must be synchronized before input to a bus master.
Parameters 1–12 are common to all transactions.

I/O Transaction Timing

Interrupt Acknowledge Timing
<table>
<thead>
<tr>
<th>Number</th>
<th>Symbol</th>
<th>Parameter</th>
<th>Min (ns)</th>
<th>Max (ns)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TsCS(AS)</td>
<td>CS To AS 1 Setup Time</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ThCS(AS)</td>
<td>AS To CS 1 Hold Time</td>
<td>60</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>TsS(AS)</td>
<td>Status To AS 1 Setup Time</td>
<td>20</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>ThS(DS)</td>
<td>Status To DS 1 Hold Time</td>
<td>60</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>TsA(AS)</td>
<td>Address To AS 1 Setup Time</td>
<td>10</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>ThA(AS)</td>
<td>Address To AS 1 Hold Time</td>
<td>50</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>TwAS</td>
<td>AS Low Width</td>
<td>70</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>TdDS(DR)</td>
<td>DS 1 To Read Data Not Valid Delay</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>TdDS(DRz)</td>
<td>DS 1 To Read Data Float Delay</td>
<td>70</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>TdAS(DS)</td>
<td>AS 1 To DS 1 Delay</td>
<td>60-2095</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>TdDS(AS)</td>
<td>DS 1 To AS 1 Delay</td>
<td>50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>ThDW(DS)</td>
<td>Write Data To DS 1 Hold Time</td>
<td>30</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**Memory Transactions**

<table>
<thead>
<tr>
<th>Number</th>
<th>Symbol</th>
<th>Parameter</th>
<th>Min (ns)</th>
<th>Max (ns)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>TdA(DR)</td>
<td>Address Required Valid To Read Data Valid Delay</td>
<td>340</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>TdAS(DR)</td>
<td>AS 1 To Read Data Valid Delay</td>
<td>230</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>TdAZ(DSR)</td>
<td>Address Float To DS (Read) 1 Delay</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>TdDSR(DR)</td>
<td>DS (Read) 1 To Read Data Valid Delay</td>
<td>95</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>TwDSR</td>
<td>DS (Read) Low Width</td>
<td>240</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>TwDSW</td>
<td>DS (Write) Low Width</td>
<td>150</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>TsDW(DSWf)</td>
<td>Write Data To DS (Write) 1 Setup Time</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>TsDW(DSWr)</td>
<td>Write Data To DS (Write) 1 Setup Time</td>
<td>210</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**I/O Transactions**

<table>
<thead>
<tr>
<th>Number</th>
<th>Symbol</th>
<th>Parameter</th>
<th>Min (ns)</th>
<th>Max (ns)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>TdA(DR)</td>
<td>Address Required Valid To Read Data Valid Delay</td>
<td>590</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>TdAS(DR)</td>
<td>AS 1 To Read Data Valid Delay</td>
<td>480</td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>TdDSI(DR)</td>
<td>DS (I/O) 1 To Read Data Valid Delay</td>
<td>255</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>TdAZ(DSI)</td>
<td>Address Float To DS (I/O) 1 Delay</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>TwDSI</td>
<td>DS (I/O) Low Width</td>
<td>390</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>TsRWR(DSI)</td>
<td>R/W (Read) To DS (I/O) 1 Setup Time</td>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>TsRWV(DSI)</td>
<td>R/W (Write) To DS (I/O) 1 Setup Time</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>TsDW(DSWf)</td>
<td>Write Data To DS (I/O) 1 Setup Time</td>
<td>30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>TsDW(DSWr)</td>
<td>Write Data To DS (I/O) 1 Setup Time</td>
<td>460</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>TdAS(W)</td>
<td>AS 1 To WAIT Valid Delay</td>
<td>195</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Interrupt-Acknowledge Transactions**

<table>
<thead>
<tr>
<th>Number</th>
<th>Symbol</th>
<th>Parameter</th>
<th>Min (ns)</th>
<th>Max (ns)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>TsIA(AS)</td>
<td>INTACK To AS 1 Setup Time</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>ThIA(AS)</td>
<td>INTACK To AS 1 Hold Time</td>
<td>250</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>TdAS(DSA)</td>
<td>AS 1 To DS (Acknowledgement) 1 Delay</td>
<td>940</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>TdDSADR(DR)</td>
<td>DS (Acknowledgement) 1 To Read Data Valid Delay</td>
<td>360</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>TwDSA</td>
<td>DS (Acknowledgement) Low Width</td>
<td>475</td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>TdAS(IEO)</td>
<td>AS 1 To IEO 1 Delay</td>
<td>3, 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>TdIEI(IEO)</td>
<td>IEI To IEO Delay</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>TdIEI(DSA)</td>
<td>IEI To DS (Acknowledgement) 1 Setup Time</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. Parameter does not apply to Interrupt-Acknowledge Transactions
2. Does not cover R/W for I/O Transactions
3. Applies only to a peripheral which is pulling INT-Low at the beginning of the Interrupt-Acknowledge Transaction
4. These parameters are device dependent: The parameters for the devices in any particular daisy chain must meet the following constraints: for any two peripherals in the daisy chain, TdAS(DSA) must be greater than the sum of TdAS(IEO) for the higher priority peripheral, TdIEI(DSA) for the lower priority peripheral, and TdIEI(IEO) for each peripheral separating them in the daisy chain.
5. The maximum value for TdAS(DS) does not apply to Interrupt-Acknowledge Transactions
6. Except where stated otherwise, maximum rise and fall times for inputs are 200 ns.
Z-bus and peripheral support packages tie distributed computer systems together

To couple support circuits to Z8000 microprocessors in an organized manner, an interconnection philosophy is needed. To this end, Zilog has developed the shared Z-bus—not a device, but a concept—to allow the construction of complex configurations of peripherals with program interfaces. This article takes the reader a step beyond basic interfacing circuits (ELECTRONIC DESIGN, Oct. 25, 1979, p. 90) and introduces both the Z-bus concept and a new family of peripheral packages, designed especially for the Z8000 µPs. Future articles will explore Z8000 software.

The Z-bus logically and efficiently organizes interconnections and transactions between Zilog's Z8000 microprocessors and their peripherals. The signals in transactions between microprocessors and peripherals inherently provide all the necessary timing, allowing asynchronous operation, so that the peripheral devices can be independent of the processor's speed and clock frequencies. In addition, the bus has a simple scheme—the daisy chain—for establishing sequential priority, as when a common system resource must be shared by several processors and peripherals.

Processors and peripherals engage in five types of transactions through the Z-bus:

- Memory transfers.
- I/O transfers
- Interrupts requests, to interrupt the Z-bus processor.
- Bus requests, to gain control of the bus for both memory and I/O transfers
- Resource requests, to gain access to a general resource.

Although memory and I/O transfers are usually between the Z-bus processor and the memory or a peripheral, some Z-bus-system peripherals, such as a direct-memory-access controller can initiate transfers after making a successful bus request.

The Z-bus system depends on strobe, request and acknowledge signals to provide the timing information between processor and peripherals (see Z-bus signal-description table). The multiplexed address/data lines, when combined with a low AS (address strobe) signal, carry the addresses of memory or internal registers within peripheral-interface packages or peripheral devices. When combined with a low DS (data strobe) the multiplexed address/data lines transfer data from or to the registers, depending on the state of the R/W (Read/Write) line.


Reprinted with permission of Electronic Design.
Data can be formatted in 8 or 16-bit groups, with memory and I/O addresses that are 16-bits long (memory addresses in the Z8001 segmented version can be as long as 24 bits).

**Peripherals get on the bus**

For a peripheral to get control of the bus, the \texttt{BUSRQ} line of the Z-bus processor in the system must be driven low. When there are several peripherals, the easiest way to generate \texttt{BUSRQ} is to wire-OR all potential bus request signals together (Fig. 1a) via the Z-bus \texttt{BRQ} line, which then becomes \texttt{BUSRQ} at the processor port.

With \texttt{BUSRQ} low after the completion of any system cycle, the processor generates a \texttt{BUSACK} low output (Fig. 1b) to acknowledge the release of the bus. At this time, all the processor outputs go into a high-impedance state to avoid affecting other signals on the bus. Meanwhile, \texttt{BUSRQ}'s low propagates through a daisy-chain hookup (Fig. 1a) among the bus requestors—the low enters each unit’s \texttt{BAI} and leaves via its \texttt{BAO} port. The device that requested control of the bus begins to use it; but the device’s \texttt{BAO} remains high, preventing lower priority bus requesters from using the bus and providing a signal that identifies it as the requestor. When the device completes its use of the bus, \texttt{BUSRQ} returns to high, followed one cycle later by a high \texttt{BUSACK} and \texttt{BAI}. This indicates that the Z-bus processor again controls the bus.

Clearly, the Z-bus processor occupies a special place on the bus, even though the processor, like any other device in the system, must wait until the bus is released before regaining control.

However, when peripherals that have intelligence and programmability approaching that of a processor share the bus, the management protocol must be more equitable than in a master-slave relationship. A protocol should be available that allows any intelligent component on the bus to seize a common resource of the system—a peripheral, memory, modem, display, etc.

**An equal-opportunity protocol**

Unlike the bus-request protocol, the resource-request chain is not dominated by a single system component. To acquire a resource, a component must issue a request signal, \texttt{MMRQ} low. All \texttt{MMRQ} signals for a given resource are wire-ORed to a common bus line (Fig. 2a). Nevertheless, the resource-requesting devices are daisy-chain connected, so that a low on the \texttt{MMRQ} line propagates through the chain—into each \texttt{MMAI} and out of each \texttt{MMAO}. However, the \texttt{MMAO} of the requesting device remains high. Thus, the combination of \texttt{MMRQ} low and \texttt{MMAO} high in a device identifies it as the temporary controller of the resource.

Before a component makes a resource request, it first checks the \texttt{MMST} (resource-status) line to see if the resource is busy. A low \texttt{MMST} line indicates busy, and additional \texttt{MMRQs} are blocked. No requestor can preempt another, but when simultaneous requests are made for the same resource, the requestor highest on the daisy chain will seize the resource first, all else being equal.

If \texttt{MMST} is high, however, \texttt{MMRQ} activates the line. After a finite delay, if \texttt{MMAI} also goes low, the resource has been seized successfully and the intended transaction can begin. Otherwise, the request is aborted—because another requester higher on the daisy chain had already seized the resource. The preempted requester may retry immediately or after some delay.

A simple logic circuit can take advantage of a requestor’s low \texttt{MMRQ} and \texttt{MMAI} and its high \texttt{MMAO} to enforce access protection for both the resource and the requestor that has successfully seized the resource.

At this point, the designer might notice that, although four lines are used on the Z-bus to control resource requests, the Z8001/8002 processors provide just two pins \texttt{M0} and \texttt{M1}. On its Multimicro Output (\texttt{M0}) pin, the \texttt{μP} issues a low signal to request a resource; its Multimicro Input (\texttt{M1}) pin tests to determine the state of the resource.

To get onto the daisy-chain with other requestors, \texttt{MMAI} and \texttt{MMAO} pins are also needed. A logic circuit, as in Fig. 2b, can provide the interface for the

![Diagram](https://via.placeholder.com/150)

**2. The resource-request chain, like the bus-request, also OR-wires the request signals, in this case MMRQ, and daisy-chains for priority (a). Several gates, however, are needed to interface a Z8000, which has just two resource-request ports, M₀ and M₁, with the daisy chain (b).**
Z8001/8002 processors: MMAI passes through gate G to MMAO as long as MO is high (not requesting the resource). While MO is high (before making a request), the state of the MMST line passes through G1 and G2 to MT. With M, high (MMST is not busy), MO can issue a request. But if another requestor higher had seized the resource first, MMAI would be low and would pass through G1 and G2 to MI to abort the μP’s request, until it could try again.

Interrupts also are daisy chained

In the interrupt protocol (as in both the bus-request scheme and the resource-request scheme), the device’s physical position in the daisy-chain—in at IEO, out at IEI—determines its priority. Also, like bus requests, interrupt requests are directed to the processor—in this case, to one of its three interrupt input ports—NMI, VI, or NVI. A separate set of interrupt-protocol signals—INT, INTACK, IEI and IEO—control each μP interrupt mode that is used. The peripheral INT ports receive the same treatment as BRQ—the INT lines for one of a processor’s interrupt modes are all wire-ORed together (Fig. 3a). The appropriate acknowledgement, decoded from the four status lines of the μP (Fig. 3b), returns via the Z-bus’ INTACK line to all the daisy-chained peripheral requestors. This procedure temporarily inhibits further interrupt requests.

Although more than one peripheral may have issued an interrupt request simultaneously, the request highest on the daisy chain prevails: Its IEO remains low, aborting any other interrupt requests further down the chain, until IEO drops low.

Three Wait cycles occur after the leading edge of INTACK to allow the daisy chain to settle (or more, if a peripheral device asks for it via the WAIT line). Then, a DS from the μP stimulates the interrupting peripheral to place its data on the bus, INTACK returns high two (or more) Wait cycles later, after completion of the transaction for which the interrupt was initiated.

After INTACK returns high, any requestor on the daisy chain can issue an interrupt; lower-priority devices are locked out until higher priority interrupts have been serviced.

I/O is main transaction

The main purpose of an interrupt request is to perform a transfer of information in or out of the processor. This I/O transaction is distinguished from every other by the μP’s status-lines code 0010, designated I/O Reference.

The bus R/W line determines the direction in which the information flows: The processor reads from the requestor device when R/W is high or writes into the device when R/W is low. Information flows via the A0 to A15 lines of the μP.

When A5 is low, the information being transferred is addresses; when A5 is low, the information is data. Word or byte formats are identified by the B/W line—word format, when low—allowing 16 or 8-bit data elements (Fig. 4).

This early-status information, which defines the transaction ahead of the actual process, allows the enabling of bidirectional drivers and other interface hardware elements. The enabling action is a distinct benefit, which simplifies interfacing peripherals.
Indeed, the Z8000 processors distinguish between I/O-transaction and memory/processor-interchange, modes only by using different status-line codes; otherwise, the two modes work almost the same way. The address/data bus, strobe lines AS and DS, and the R/W, B/W, and N/S lines are shared by both I/O and memory transactions; therefore the interface buffers can be shared by substantially fewer processor pins.

One difference in the modes—an extended address capability to 23 bits—applies only to memory, when the segmented Z8001 version of the processor is used.

Memory is organized into two 8-bit-wide banks. One bank contains the most-significant bytes of the addressable words; the other contains the lower bytes. The banks can be activated together or separately by a B/W low signal (Fig. 4).

Memory and I/O functions must then be done sequentially, but the high-speed of the processor transactions can handle most applications adequately. If necessary, the WAIT line can be called upon to extend a transaction for I/O and memory, because the device (or memory) is not ready or cannot work fast enough to keep up with the processor.

**Help for the busy processor**

When the processor gets too busy to handle all its peripherals efficiently, then Zilog's Universal Peripheral Controller (Z-UPC), one of several support packages that will soon be available, can step in and help out (Fig. 5). With pin functions AS and DS, R/W and WAIT, IEI, and IEO, INT and INTACK, Z-UPC can plug right into the Z-bus and serve as a complete slave microcomputer for distributed processing. It can:

- Control peripheral devices with internal ROM or RAM instructions.
- Manipulate data arithmetically or format buffer data in internal registers.

Based on the Z8 microprocessor architecture and instruction set, the Z-UPC is an intelligent device that can unburden the main processor and greatly increase

---

**Z-Bus signal descriptions**

- **AD0-AD15** Address/Data Lines. The multiplexed address/data lines are used for both I/O and memory transfers.
- **AS** Address Strobe. The rising edge of AS indicates addresses are valid.
- **BRO, BAI, BAO** Bus Request, Bus Acknowledge Input, Bus Acknowledge Output. Other Z-Bus masters, such as the Z-DMA, use this bus control request chain to take control from the CPU.
- **DS** Data Strobe. DS times the data in and out of the CPU.
- **EXEENDED ADDRESSES** The number, type and nature of these lines depend on the CPU used.
- **INT, INTACK, IEI, IEO** Interrupt, Interrupt Acknowledge, Interrupt Enable Input, Interrupt Enable Output. This set of lines is used for interrupt control and the interrupt daisy chain for each type of interrupt.
- **RESET** Reset. A Low on this line resets the system.
- **R/W** Read/Write. R/W indicates the CPU is reading or writing.
- **Status Lines** The status lines distinguish the different kinds of bus transactions, such as I/O or memory.
- **WAIT** Wait. This line indicates to the bus controller that the responder is not ready for data transfer.
- **MMST, MMRO** Multi-Micro Status, Multi-Micro Request, Multi-Micro Acknowledge Output, Multi-Micro Acknowledge Input. This resource-request chain controls access to common resources.

---

4. The byte/word and read/write organization of words is handled almost the same way for I/O transactions between peripherals and processor as for I/O transactions between memory and processors.
overall system efficiency and speed. It generates almost any control signal that a peripheral device might need. Operating on the same 4-MHz clock as the Z8000 µPs, it executes instructions in an average of just 2 µs.

Not only speed, but flexibility is attained. An extensive register file of 256 byte-registers, organized into 16 groups of 16 working registers each make the Z-UPC very versatile. Short-format instructions expedite the access to any group. The file includes 234 general-purpose, 19 status-and-control (including two 16-bit counter/timer) and three I/O-port registers. Add six levels of priority interrupts and the Z-UPC is indeed a flexible support package.

Any general-purpose register can be used as an accumulator, address pointer, index register or stack for the Z-UPC's program. All unused general-purpose registers can then act as data buffers between the master processor and the peripheral device. In addition, communications between the master processor and the Z-UPC takes place via one of the groups of 16 registers, which are accessed directly by the master processor over the Z-bus Address/Data (ADx) lines.

**Examining the I/O ports**

The Z-UPC's three I/O ports also allow great flexibility. Two of the I/O ports are 8-bits each; the third has 8 bits for I/O that can be shared between I/O and control lines, as determined by the program. In fact, all the I/O ports can be programmed in many combinations as input, output or bidirectional lines, with or without a handshake protocol.

When its P3o, P3b, P3c, and P 37 pins are programmed as IE1/IE0, INTACK and INT lines, the Z-UPC fits easily into the Z-bus's daisy-chain priority system. As an alternative, the controller can be programmed to operate with a polled system—a concept that is also compatible with the Z-bus.

Not all peripheral interfacing tasks need all the intelligence and flexibility that the Z-UPC possesses. Zilog's Counter/Timer and Parallel I/O (Z-CIO), with its two independent 8-bit bidirectional I/O ports and special-purpose 4-bit I/O port, can satisfy most ordinary needs for parallel I/O interfacing and counting/timing (Fig. 6).

Either of the Z-CIO's two identical 8-bit I/O ports can operate in a handshake-byte or bit-by-bit mode.

---

6. By taking care of parallel I/O interfacing and counting, the Z-CIO peripheral-interfacing circuit chip can remove a heavy burden from the processor in a complex Z8000 system.

---

5. A universal peripheral controller (Z-UPC) can take a great amount of the load off a microprocessor, especially when the processor is interfacing peripheral devices that demand a lot of detailed attention.
7. For long-distance serial communications with a processor, the Z-SCC converts parallel-to-serial data and then serial-to-parallel for either synchronous or asynchronous data links.

In the later mode, the direction of each bit can be individually programmed. Like the universal controller, the two ports can perform in the handshake mode, as inputs, outputs or bidirectional lines; also, they can be linked into one 16-bit port. In addition, each of the 8-bit ports includes pattern-recognition logic to generate an interrupt when a specified pattern is detected.

Four handshake protocols are available: the IEEE-488, an interlocked (with another Z-CIO or Z-UPC), a strobed and a pulsed.

The pulsed handshake connects one of the Z-CIO's counters with logic, to interface a mechanical device such as a printer. The special-purpose 4-bit I/O port provides the handshake controls: a Wait/Request line for high-speed data transfer or general-purpose I/O. The programming and status for all the control features reside in 12 registers provided for each port.

The Z-CIO's three, independent, identical 16-bit counter/timers (two of the counters can be programmed to form a 32-bit counter/timer) can help to control a device. Each counter/timer consists of a 16-bit down-counter and four registers as follows: a 16-bit register, to hold the initial value (called the Time Constant), which is loaded into the down-counter; another 16-bit register, to hold a current down-count output, when strobed; and two 8-bit registers to hold mode, control and status information.

Either the counting or the timing function can be programmed for single-cycle (one-shot) or continuous operation with a pulse or square-wave output. Up to four control lines—for each counter/timer—can act as the counter input, enable input, trigger input and counter/timer output, as required.

Whether counting or parallel interfacing, the Z-CIO can substantially unburden the master processor in a computer system, especially when complex peripherals demanding high service must be handled. The Z-CIO is also fully compatible with the Z-bus and provides the full complement of bus control signals and daisy-chain priority pins (IEI/IEO).

Serial unit supports many protocols

Also supporting the Z-bus family is the Z-SCC, Serial Communications Controller, a peripheral-interfacing package for serial communications or data-transfer applications (for example, with disks and cassettes). The package contains two independent full-duplex channels, each with its own quartz-crystal oscillator, baud-rate generator and digital phase-locked loop for clock recovery (to 1 Mbit/s). Each channel also provides facilities for modem/control (Fig. 7a).

The Z-SCC is programmable for NRZ, NRZI or FM-data encoding. A channel in an asynchronous mode can operate with 5 to 8-bit codes per character plus 1, 1-1/2 or 2 bits per character as stop bits. In addition the package provides such features as break detection and generation, and parity, overrun and framing error detection.

In the synchronous mode, the Z-SCC handles such protocols as IBM Bisync or bit-oriented HDLC and

8. The Z-FIO general-purpose bidirectional buffer can interconnect devices operating at different speeds. It is not limited to Z8000 configurations, but can handle almost any general-purpose μP system.
SDLC with frame-level control, automatic zero-insertion and deletion, I-field residue handling, abort generation and detection, CRC generation and checking and loop-mode operation. Parity and overrun features also apply to synchronous operation.

Fig. 7b shows one of the Z-SCC's channels connected as a synchronous data-link—the loop (SDLC) mode. Note the absence of clock lines. With NRZI or FM data, no clock lines are needed, since the clock can be recovered at the receive end from the bit stream by the Z-SCC's digital phase-locked loop. The other channel, via a modem under control of the Z-SCC, is shown servicing an asynchronous serial port.

Basically the Z-SCC functions as a parallel-to-serial and serial-to-parallel converter, but it does more: Its sophisticated repertoire of internal functions greatly reduces the amount of external supporting logic needed for a wide variety of serial-communications applications in distributed-processing systems.

Another great saver of external assorted logic in distributed-processor operation is the Z-FIO general-purpose bidirectional buffer.

First-in, first-out

The Z-FIO can interconnect components or subsystems (of almost any μP including the Z8000) operating at different speeds. It can accept 128 bytes of data, which it then holds until they are called for by another device in the system. In this way, interrupt servicing time can be cut two orders of magnitude in most I/O transactions. Moreover, the capability of moving variable-sized blocks under either direct-memory access or interrupt control greatly facilitates system throughput, which is especially important with fast peripheral circuits.

The internal functions of the Z-FIO are shown in Fig. 8. Its two sets of Address/Data ports are identical except for programming. The A set (programmed by pins M₀ and M₁) and the B set (programmed by bits SL₀ and SL₁) have in common a 128 × 128 RAM for data storage, two 7-bit counters and several registers. The RAM can read and write both simultaneously and independently: The A set can write a byte of data into the RAM without disturbing a simultaneous read operation at the B set. The counters address the RAM and, by means of a subtractor, determine the number of bytes remaining in the memory. This number can be read from a status register dedicated to each set.

When compared internally with the memory-status register, a programmable register generates an interrupt for starting and stopping DMA transfers. Another pair of registers permits direct communication between the ports by bypassing the main buffer memory.■
AN OPTIMISING DRIVER
FOR NEC SPINWRITER
AND DIABLO PRINTERS
An Optimising Driver
For NEC Spinwriter
And Diablo Printers

CONTENTS

1 INTRODUCTION .......................... 6-6

2 THE DRIVER FROM A NON-TECHNICAL USER'S VIEWPOINT 6-7
   2.1 Loading and initialising the driver 6-7
   2.2 Using the driver 6-8
   2.3 Automatic Formfeed 6-9
   2.4 Use of Cut Sheets 6-10
   2.5 Character Attributes 6-10

3 THE DRIVER FROM A TECHNICAL USER'S VIEWPOINT 6-12
   3.1 The Interface Hardware 6-12
   3.2 The Driver Software - General 6-14
   3.3 The Modules 6-15

4 CONCLUSION .......................... 6-17

5 APPENDIX - MODULE LISTINGS 6-18
   PRINTER.DRIVER.0 6-18
   PRINTER.DRIVER.1 6-25
   PRINTER.DRIVER.2 6-30
   SPINWRITER 6-34
   DIABLO 6-42
   CALL.SYSTEM 6-49
   RIO.IOINTERFACE 6-50
   SIB 6-52
   PLZ_INTERFACE_MACRO 6-57
1 INTRODUCTION

This printer driver was written to suit a variety of applications, but in particular, it was prepared to be used as a companion to the Zilog text formatting utility, ZFORM.

It is also well suited to applications requiring the printing of long lines, such as are found in connection with business orientated data base management programs.

It provides compatibility with both of the most commonly used high performance, high quality printers, the NEC SPINWRITER, and the DIABLO.

The features provided by the driver ensure that the printer operates at its maximum possible speed under all circumstances. Adjacent spaces and tabs are merged into single head movements, and the printing direction is fully optimised to minimise unnecessary head travel. This has the effect not only of raising print speeds, but of considerably reducing noise and vibration, although this effect will not be obvious unless two printers are operating near to each other under different control algorithms.

Several further features are controlled by "attributes" which could be easily extended, but as currently implemented allow for underlining, BOLD character printing, RED printing, superscripting, and subscripting, or any combination of these features, which can even be invoked at the character level within words.

The driver can optionally skip to top-of-page at a given page line count, to prevent printing over fold lines in continuous stationery, and, if required will allow the operator to load a fresh sheet of cut paper after ejecting the current "page". In this case, a message is sent to the operator console, together with a bell-code, as indications that operator intervention is required. When the new sheet has been loaded, the operator presses a key on the console in order to continue printing. In this situation, the operator is able to give a specific response in order to inform the driver that pauses are no longer needed between pages. This response is given by pressing either upper-case, or lower-case C (for Continue).

The facility for using cut sheets is essential in the context of preparing documents such as letters, which will usually be printed on headed paper for the first sheet, and plain paper for continuations.

The printing operation may be aborted at any time by hitting the <ESC> key on the console, and may be halted temporarily at any time by raising the cover on the printer itself.

If the printer is equipped with a detector for end-of-ribbon, which is a standard feature of the SPINWRITER, it will automatically pause for the operator to load a new cartridge. Printing will be suspended, and an audible alarm given to indicate that operator
intervention is required. After the cartridge has been replaced, printing resumes without any visible discontinuity.

2 THE DRIVER FROM A NON TECHNICAL USER'S VIEWPOINT

2.1 LOADING AND INITIALIZING THE DRIVER

The driver is loaded by RIO, the operating system executive, either in response to a direct command from the operator entered at the console keyboard, or to a command included in a file containing commands, such as the file OS.INIT.

In either case, the operation involves the use of the RIO command ACTIVATE.

Assuming that a NEC Spinwriter is in use, and the files of the disc supplied have been used without modification, the actual device driver will be known as $NEC. The command is therefore

```
%ACTIVATE $NEC
```

Optionally, the operator may make the printer driver known by a 'Logical Unit Number', preferably 3, which is used by convention for printers. This can be achieved by the additional command:-

```
%DEFINE 3 $NEC
```

There are several RIO utilities, such as PRINT, and CAT, which automatically send their output to whatever device has been associated with LUN=3, and it is recommended that the procedure given above be used. This adds significantly to the general convenience of using the system.

When the driver receives an Initialisation request from RIO, it performs some general housekeeping operations, such as preparing the electrical characteristics of the hardware interface to the printer, and placing the printhead in a known position, and then it sends a message to the operator, asking whether it is to operate in the manner needed if cut sheets of paper are to be used, and whether it is to automatically perform paper throws to prevent printing over folds in continuous stationery. These two functions are separately controllable, as some programs which use the printer driver perform similar functions themselves. In that case, irritating interactions could possibly occur, resulting for example, in alternate pages being
Spinwriter/Diablo driver

completely blank. Initialisation requests are sent to the printer driver whenever it is ACTIVATEd, or, if required, can be issued specifically on demand by the operator by using the command:-

```
%I $NEC
```

RIO prompt

Operator command

In either event, the driver will introduce itself by a message on the system console , giving its revision level, and will then ask the operator two questions. The responses are entered on the console keyboard as single characters. The response character Y in either upper, or lower case, specifies 'YES' to the question.

The dialogue takes the following form:-

```
%ACTIVATE $NEC

Printer Driver rev. 2.1

Cut sheets ?  Y
Auto formfeed ? Y

%
```

In the above example, the two responses for Yes are shown. Any other character response apart from Y is interpreted as NO.

If the requirements of the driver change during a session, all that is needed to redefine the characteristics is for the operator to give the command to re-initialise the driver, as exampled previously. The two questions will be re-asked. If this is done, it does NOT alter the previously defined association between the driver and a RIO LUN.

2.2 USING THE DRIVER

RIO can be told to pass data ( ie. text ) to the printer in essentially two different ways. One is much simpler to use than the other, and relies on having DEFINEd that the driver is known to RIO as LUN=3.

Assuming that a file is known to exist, such as PRINTER.DRIVER.MANUAL and that the printer driver is indeed known as LUN=3, all that the operator needs to enter is:-

```
%PRINT PRINTER.DRIVER.MANUAL
```

and the contents of that file will be printed. Obviously the text actually appearing on the paper will largely reflect exactly what is contained in the file, but pagination can be affected by whether the operator has selected automatic formfeed. The use of cut sheets does
not affect the printed output, merely the type of paper stock which can conveniently be used.

Alternatively it is often possible to send text direct to $NEC as it is being generated, instead perhaps, of preparing a file which would have to be printed subsequently.

This can be achieved in different ways depending on the program which is generating the text. As an example, we will show the use of ZFORM.

ZFORM outputs its formatted text to the system console unless the operator specifies an alternative. This is particularly convenient, as most operators would require to view the fully formatted text on their VDU more frequently than actually printing it.

An alternative is specified by giving an "O=" option, which can define either the name of a file which is to be prepared to contain the formatted output text, or a driver, such as $NEC. In this case, the formatted text would be sent directly to $NEC, and therefore to the printer, as it is being generated.

For example:

```
%ZFORM PRINTER.DRIVER.MANUAL O=$NEC
```

### 2.3 Automatic Formfeed

If the operator has selected the automatic formfeed option, after printing a given number of lines on a page, the driver will tell the printer to perform a paper-throw, i.e. formfeed operation, which prepares it to start printing on a new sheet.

The driver maintains a count of the number of lines it has printed since the last time it started a new sheet, and when it reaches 63 (this can be varied if necessary) it requests a formfeed. As most paper sheets have a length equal to 66 lines, this means that every page would have at least three blank lines, and they would normally be positioned to bracket the folds in continuous stationery.

However, if the driver finds a formfeed code in the text being printed, it requests a paper throw, and zeros its line counter.

Most files of text which are prepared by Zilog utility programs do contain embedded formfeed codes, and it is for this reason that the driver usually does not need to insert any automatically. Interaction could occur if the text being printed were pagenated using the same line count per page as the printer driver. Blank alternate pages would result.

It is very useful, however, to be able to neatly print, and pagenate, files which are prepared directly by an operator using a text editor,
such as input files for use by ZFORM for example, and it is then that
the auto formfeed option is likely to be used.

2.4 USE OF CUT SHEETS

If the operator has selected the cut sheet option, whenever the
driver has sent a paper throw instruction to the printer, which will
cause the current sheet to be ejected, it will then cause the
console's buzzer to be sounded and send a message to the operator at
the system console. The message represents a request to load a new
sheet of paper, and then to press a key on the keyboard as an
instruction to the driver that printing can resume. If any key other
than either an upper case, or a lower case C is pressed, the driver
will continue to request the loading of fresh sheets whenever it has
ejected the current sheet. However, if the letter C is used in
response, printing will continue in the expectation that continuous
stationery is then in use.

2.5 CHARACTER ATTRIBUTES

As implemented herein, the attributes of BOLD, UNDERLINE, RED,
SUPERSCRIPT, and SUBSCRIPT, are invoked by two-character "ESCAPE code
sequences" in the text being printed, employing codes which do not
correspond to printable characters.

This technique ensures that text files which contain the necessary
control codes for these functions can be printed by printers such as
TALLY, CENTRONICS etc. and by conventional Visual Display Units when
handled by standard Zilog drivers, with total compatibility.
Obviously these peripherals cannot produce the effects obtainable
when using a Spinwriter, but the text format and content would be the
same. It would perhaps be regarded as a draft quality output which
can be prepared at very high speed.

The attribute control sequences are as follows:

BOLD <ESC> CTRL-B
UNDERLINE <ESC> CTRL-U
RED <ESC> CTRL-R

POSITIVE HALF-LINEFEED ( for SUBSCRIPTING ) <ESC> CTRL-F
NEGATIVE HALF-LINEFEED ( for SUPERSCRIPTING ) <ESC> CTRL-N

The first three attribute control sequences operate in the manner of
'toggles'. i.e. if, for example, the printer is outputting in black, then the sequence <ESC> CTRL-R would switch it to red, and a further
<ESC> CTRL-R would switch it back to black again.

The receipt of a formfeed request, whether internally generated, or
received as part of the text being printed, cancels out any current

6-10
superscripting or subscripting. i.e. the top line of a new page will always start in the same place relative to the top of the paper.

When preparing an original text file, the method for embedding the control code sequences will depend on exactly what software utility is being used at that time, however, as an example, we will illustrate the use of the standard RIO TEXT EDITOR.

As this editor makes a special use of the code generated by depressing the 'ESCAPE' key on the console keyboard, the following example illustrates perhaps the most involved way of incorporating attribute control sequences into the text.

The editor can be told to pass the code from the ESCAPE key into its output file by preceding it with the key labeled '\'. This prevents the editor from interpreting the code from ESCAPE for its special function.

Now, let's see exactly what keystroke sequences would have to be used in order to print the following line of text:-

This illustrates **BOLD** printing.

Using the terminology that <ESC> means the ESCAPE key, **CTRL-B** means pressing the 'B' key whilst holding down the CONTROL-SHIFT key, the operator would use the following key sequence:-

This illustrates\<ESC>CTRL-B BOLD\<ESC>CTRL-B printing.

Notice that that there must not be any character between the <ESC> and the following attribute control code.

Character attributes can be individually controlled. The inclusion of an attribute control sequence within the text is really interpreted as an instruction to make use of the currently set attribute(s) until redefined.
3 THE DRIVER FROM A TECHNICAL USER'S VIEWPOINT

3.1 THE INTERFACE HARDWARE

The printer is interfaced to the host system by the use of one of the four serial channels provided by an SIB (Serial Interface Board).

There is one other major software utility which currently employs an SIB channel, i.e. the Asynchronous Communications Package.

The printer driver and the communications package are totally compatible with each other, and can successfully co-operate within a system.

This driver assumes the use of channel 2 of the SIB, which is installed in an MCZ-1/nn style Zilog system without modification. Channels 2 and 3 are pre-wired in all systems currently shipped. The communications package uses channel 3.

SIB modules have a great many link areas enabling the characteristics of each channel to be tailored to precisely suit the user's needs. The following link definitions are specific to channel 2, and the printer driver when operating with either a NEC Spinwriter model 5510/5520, or a DIABLO model 1610/1620.

Clock distribution:-

J3-6 to J3-12
J2-3 to J2-15
J2-3 to J2-16

Connections between USART-2 and the printer:-

J7-1 to J7-13
J7-2 to J7-14
J7-3 to J7-11
J7-4 to J7-12
J7-5 to J7-10
J7-6 to J7-9

The above links configure channel 2 of the SIB to communicate with a 'terminal', and interface the following signals at the 25 way socket with which the printer is to be connected. In most MCZ-1/nn systems this is labeled 'J102'.
The actual interface signals are as follows:-

J102-7  Ground
J102-8  'spare'
J102-5  Clear To Send -> To printer
J102-6  Data Set Ready -> To printer
J102-20 Data Terminal Ready <- From printer
J102-4  Request To Send <- From printer
J102-3  Received data <- From printer
J102-2  Transmitted data -> To printer

The printer driver assumes that a rate of 1200 bauds will be used, and switches in the printer must be appropriately set. As those settings depend upon the exact printer model number, it is impracticable to give them here.

All other links on the SIB are either to be left as when delivered, or set as required for defining the characteristics of the channels which are not used by the printer driver.
3.2 THE DRIVER SOFTWARE - GENERAL

This driver operates by placing character codes into a line buffer in locations which correspond to columns of the output line. ie. the content of the line buffer is columnated.

Before being written into, the line buffer is cleared to contain space codes in bits 0 -> 6. Bit 7 is handled independently, and, when set, defines the location of a tab-stop. Each time the line buffer is cleared, the setting of bit 7 is left unaffected in each location.

The tab bits are defined after a call is made to determine the current tab locations in use by the system console driver. This is done each time an initialisation request is received by the printer driver.

A second columnated buffer is used in addition to the line buffer. This contains character attributes and effectively extends each character code to include bits which independently define whether the character is to be printed in red, bold, underlined, or as a superscript or subscript. As currently implemented, there are three spare attribute bits, which could easily be allocated for specific extensions to the driver's capabilities.

The attribute buffer is loaded according to the attribute control sequences embedded in the input text. These are used to directly control the value of the variable NEXT_ATTRIBUTE, which is copied into the attribute buffer when a character is placed into the line buffer.

After the line and attribute buffers have been loaded, the driver decides whether the current printhead position is nearer to the left or right end of the line about to be printed, and is therefore able to perform an absolute tab to the nearer end, and output the line in the appropriate direction.

Notice that the driver never issues a carriage-return code to the printer. It always sends absolute tabs and linefeeds. This is due to the danger of accidentally locking the "Auto Linefeed" switch of some printers, which is sometimes located near to frequently used controls. Its use would destroy carefully defined formats.

It is expected that any programmer who wishes to understand, or modify, the driver will be able to do so easily after reading the module listings. Therefore a blow-by-blow descriptions of the operation is considered unnecessary.
3.3 THE MODULES

The driver software consists of a number of modules, each being written in the language chosen to be most appropriate for the function it performs.

The modules perform the following general functions:

<table>
<thead>
<tr>
<th>MODULE</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRINTER.DRIVER.0 (PLZSYS)</td>
<td>Receives the RIO request vector from RIO.IO INTERFACE and interprets the request code. Makes a call back to RIO in order to determine the TABSTOP locations within the standard RIO console driver so as to be able to use the same locations itself.</td>
</tr>
<tr>
<td>PRINTER.DRIVER.1 (PLZSYS)</td>
<td>Contains the procedures for building the LINE and ATTRIBUTE buffers.</td>
</tr>
<tr>
<td>PRINTER.DRIVER.2 (PLZSYS)</td>
<td>Contains the procedures for optimising print direction and removing data from the LINE and ATTRIBUTE buffers during actual printing.</td>
</tr>
<tr>
<td>DIABLO and SPINWRITER (PLZSYS)</td>
<td>These modules contain printer-dependent procedures for selecting print direction, absolute tabbing, selecting ribbon colour, requesting positive or negative half linefeeds for subscripting and superscripting, and management of ETX/ACK protocol for maintaining control of the buffer in the printer itself. This is the only module of the driver which is not common to both the Spinwriter and the Diablo.</td>
</tr>
</tbody>
</table>
This very simple module merely converts the IY register content received from RIO as the request-vector-pointer into a PLZSYS procedure parameter. It then passes control to the main procedure in PRINTER.DRIVER.O.

Return to RIO is through this module so that IY can be restored.

This module contains the routines to set up the basic I/O interface to the printer. All routines may be called direct from PLZSYS code.

All character level I/O is performed by this module.

This module receives a pointer as a parameter from a PLZSYS program, and passes it as an RIO I/O request vector pointer to RIO. In this driver it is used when requesting the status of $CON to determine if an abort has been requested, for getting tab locations from $CON, also for issuing messages to, and obtaining operator responses from $CON.

For convenience only.
CONCLUSION

Hopefully this note will have given the reader a few ideas about the use of PLZSYS in association with Assembly Language for I/O driver writing. The author cannot realistically recommend its use if memory space is at a premium, but certainly does recommend it wholeheartedly if an objective is to produce intelligible, easily adaptable code quickly. The entire driver described herein took less than 30 man-hours to design, implement and test.

The source code files for all modules are available from Zilog's franchised distributors as part of the software library.

Any users' improvements to this driver would be warmly welcomed if contributed to the Software Library.
PLZSYS 2.02

1  PRINTING_DRIVER_0  MODULE
2
3  !  Extended 3/3/79 to allow for subscripting and superscripting!
4  !  Also for operator controlled page-waits and auto formfeed!
5
6
7  TYPE
8
9  RIO_REQUEST_VECTOR  RECORD [ LUN  BYTE
10     REQ  BYTE
11     DTA  BYTE
12     DTL  WORD
13     CRA  WORD
14     ERA  WORD
15     CCOD  BYTE
16     SPV_ADD  WORD ]
17
18  CONSTANT
19
20     INITIALISE := 0
21     ASSIGN  := $02
22     OPEN  := $04
23     CLOSE := $06
24     WRITE_BINARY := $0E
25     WRITE_LINE  := $10
26     READ_LINE := $0C
27     READ_STATUS := $40
28     WRITE_STATUS := $42
29     DEACTIVATE := $44
30     INVALID_OPERATION_REQUEST := $C1
31     PROGRAMME_ABORT := $49
32     GOOD_RETURN := $80
33     CONIN := 1
34     CONOUT := 2
35     ASCII_SPACE := ' '
36     ASCII_CR := 'R'
37     ASCII_LF := 'L'
38     ASCII_FF := 'P'
39     ASCII_BELL := $07
40     BLACK := $01
41     TRUE := 1
42     FALSE := 0
43     INTERRUPT_REQUEST_MASK := $FE
44     TAB_BIT := $80
45
46  EXTERNAL
47
48     REQUEST_BLACK  PROCEDURE
49     GET_CODE  PROCEDURE
50     SETCH2  PROCEDURE
51     ABSOLUTE_TAB  PROCEDURE ( BYTE )
52     FORMFEED  PROCEDURE
Appendix

6-19

PRINT_LINE_BUFFER
CALRIO
CLEAR_LINE_BUFFER
LINE_CONTAINS_PRINTABLE_CHAR
CODE
ATTRIBUTE_SEQUENCE_FLAG
NEXT_ATTRIBUTE
LINE_FINISHED_FLAG
EOF_FLAG
BYTES_TAKEN_FROM_SOURCE
BYTE_COUNT
CONSOLE_STATUS_BUFFER
LINE_BUFFER
LINE_BUFFER_PTR
INTERNAL

INPUT_CHAR

date_code
NEW_SHEET_MSG
BELL_STRING
CUT_SHT_QUES
AUTO_FM_FEED_QUES
NL_ARRAY

GENERAL_RIO_CALL_VECTOR

GLOBAL

REQUEST_CODE
SOURCE_PTR
DATA_LENGTH
ABORT_FLAG
AUTO_FF_FLAG
PAGE_WAIT_FLAG

CALL_RIO

PROCEDURE

PROCEDURE ( ^BYTE )

RETURNS ( BYTE )

PROCEDURE

BYTE
BYTE
BYTE
BYTE
BYTE
WORD
BYTE
ARRAY [ 5 BYTE ]
ARRAY [ 163 BYTE ]
^BYTE
BYTE
RIO_REQUEST_VECTOR

BYTE
^BYTE
WORD
BYTE
BYTE
BYTE
BYTE
UNIT
REQUEST
DATA_ADDRESS
DATA_LENGTH

UNIT
REQUEST
DATA_ADDRESS
DATA_LENGTH

^BYTE
WORD

RETURN_CODE BYTE
ENTRY

GENERAL_RIO_CALL_VECTOR.LUN := UNIT
GENERAL_RIO_CALL_VECTOR.REQ := REQUEST
GENERAL_RIO_CALL_VECTOR.DTA := DATA_ADDRESS
GENERAL_RIO_CALL_VECTOR.DTL := DATA_LENGTH

'%'RPrinter Driver rev. 2.1%'R'
'Load new sheet, hit a key:'
ASCII_BELL
'Cut sheets?'
'Auto formfeed?'
'%'R'

PRINTER.DRIVER.0
RETURN_CODE := CALRIO ( #GENERAL_RIO_CALL_VECTOR.LUN )

IF RETURN_CODE <> GOOD_RETURN
    ABORT_FLAG := TRUE
FI

END CALL_RIO

MAYBE_ABORT PROCEDURE
ENTRY
CALL_RIO ( CONIN
    READ_STATUS
    #CONSOLE_STATUS_BUFFER[0]
    1 )

IF ( CONSOLE_STATUS_BUFFER[0] AND %20 ) = 0
    ABORT_FLAG := TRUE
FI
END MAYBE_ABORT

NEWLINE PROCEDURE
ENTRY
CALL_RIO ( CONOUT
    WRITE_BINARY
    #NL_ARRAY[0]
    SIZEOF NL_ARRAY )
END NEWLINE

GET_CHAR PROCEDURE
ENTRY
CALL_RIO ( CONIN
    READ_LINE
    #INPUT_CHAR
    1 )

IF INPUT_CHAR <> '%R' THEN NEWLINE FI
END GET_CHAR

PAGE_WAIT PROCEDURE
ENTRY
CALL_RIO ( CONOUT
WRITE_BINARY
#NEW SHEET_MSG[0]
SIZEOF NEW SHEET_MSG + SIZEOF BELL_STRING )
GET_CHAR
IF INPUT_CHAR
CASE 'C' 'c'
THEN
PAGE_WAIT_FLAG := FALSE
NEWLINE
FI
END PAGE_WAIT

GET_FLAGS PROCEDURE
ENTRY
CALL_RIO ( CONOUT
WRITE_BINARY
#date_code[0]
SIZEOF date_code )
CALL_RIO ( CONOUT
WRITE_BINARY
#CUT_SHT_QUES[0]
SIZEOF CUT_SHT_QUES )
GET_CHAR
PAGE_WAIT_FLAG := FALSE
IF INPUT_CHAR
CASE 'Y' 'y'
THEN PAGE_WAIT_FLAG := TRUE
FI
CALL_RIO ( CONOUT
WRITE_BINARY
#AUTO_FM_FEED_QUES[0]
SIZEOF AUTO_FM_FEED_QUES )
GET_CHAR
AUTO_FF_FLAG := FALSE
IF INPUT_CHAR
CASE 'Y' 'y'
THEN AUTO_FF_FLAG := TRUE
FI
NEWLINE
NEWLINE
END GET_FLAGS

GET_TAB_LOCATIONS PROCEDURE

LOCAL COUNTER BYTE

ENTRY

CALL_RIO ( CONIN
READ_STATUS
#CONSOLE_STATUS_BUFFER[0]
139 )

COUNTER := 0
LINE_BUFFER_PTR := # LINE_BUFFER [0]
DO
IF COUNTER = 163 THEN EXIT FI
IF COUNTER < 134
THEN
IF LINE_BUFFER_PTR <> 0
THEN
LINE_BUFFER_PTR := TAB_BIT
ELSE
LINE_BUFFER_PTR := 0
FI
ELSE
LINE_BUFFER_PTR := TAB_BIT
FI
LINE_BUFFER_PTR := INC LINE_BUFFER_PTR
COUNTER += 1
OD
CLEAR_LINE_BUFFER

END GET_TAB_LOCATIONS

EJECT_PAGE PROCEDURE

ENTRY

IF PAGE_WAIT_FLAG = TRUE
THEN
PAGE_WAIT_FLAG := FALSE
FORMFEED
PAGE_WAIT_FLAG := TRUE
ELSE
FORMFEED
FI

END EJECT_PAGE
Appendix

PLZDVR

PROCEDURE (VECTOR_PTR^RIO_REQUEST_VECTOR)

ENTRY

REQUEST_CODE := VECTOR_PTR^REQ AND INTERRUPT_REQUEST_MASK
SOURCE_PTR := VECTOR_PTR^DTA
DATA_LENGTH := VECTOR_PTR^DTL

VECTOR_PTR^CCOD := GOOD_RETURN
VECTOR_PTR^DTL := 0

BYTES_TAKEN_FROM_SOURCE := 0
EOF_FLAG := FALSE
ABORT_FLAG := FALSE

IF REQUEST_CODE

CASE INITIALISE

THEN

SETCH2
BYTE_COUNT := 0
ABSOLUTE_TAB (1)
EJECT_PAGE
ATTRIBUTE_SEQUENCE_FLAG := FALSE
NEXT_ATTRIBUTE := BLACK
REQUEST_BLACK
GET_TAB_LOCATIONS
GET_FLAGS

RETURN

CASE ASSIGN

THEN

RETURN

CASE OPEN

THEN

GET_TAB_LOCATIONS
RETURN

CASE CLOSE,DEACTIVATE

THEN

ABSOLUTE_TAB (1)
EJECT_PAGE
RETURN

CASE WRITE_BINARY

THEN

DO

IF DATA_LENGTH = BYTES_TAKEN_FROM_SOURCE

THEN

EXIT
FI
GET_CODE
IF EOF_FLAG = TRUE
    EXIT
FI
IF ABORT_FLAG = TRUE
    VECTOR_PTR^.CCOD := PROGRAMME_ABORT
    EJECT_PAGE
    EXIT
FI
OD
VECTOR_PTR^.DTL :=BYTES_TAKEN_FROM_SOURCE
RETURN
CASE WRITE_LINE
THEN
    LINE_CONTAINS_PRINTABLE_CHAR := FALSE
    DO
        IF DATA_LENGTH =BYTES_TAKEN_FROM_SOURCE
            THEN
                PRINT_LINE_BUFFER
                EXIT
        FI
        GET_CODE
        IF CODE = ASCII_CR
            THEN EXIT
        FI
        IF ABORT_FLAG = TRUE
            THEN
                VECTOR_PTR^.CCOD := PROGRAMME_ABORT
                EJECT_PAGE
                EXIT
        FI
    OD
VECTOR_PTR^.DTL :=BYTES_TAKEN_FROM_SOURCE
RETURN
ELSE
    VECTOR_PTR^.CCOD := INVALID_OPERATION_REQUEST
FI
END PLZDVR
END OF ZCODE GENERATION
0 ERROR(S) 0 WARNING(S)
Spinwriter/Diablo driver

PLZSYS 2.02

1       PRINTER_DRIVER_1          MODULE
2
3   ! Extended 3/3/79 to allow for subscripting and superscripting
4
5
6       CONSTANT
7
8       TRUE                      := 1
9       FALSE                     := 0
10
11      ASCII_SPACE               := ' '.
12      ASCII_TAB                  := %09
13      ASCII_BS                   := %08
14      ASCII_ESC                  := %1B
15      ASCII_FF                   := %0C
16      ASCII_CR                   := '%R'
17      ASCII_LF                   := '%L'
18      ASCII_CONTROL_R           := %12 ! COLOUR CHANGE !
19      ASCII_CONTROL_B           := %02 ! BOLD !
20      ASCII_CONTROL_U           := %15 ! UNDERLINE !
21      ASCII_CONTROL_N           := %0E ! SUPERSCRIPT !
22      ASCII_CONTROL_F           := %06 ! SUBSCRIPT !
23
24      BLACK                      := %01
25      RED                        := NOT BLACK
26      BOLD                       := %02
27      NOT_BOLD                   := NOT BOLD
28      UNDERLINE                  := %04
29      NOT_UNDERLINE             := NOT UNDERLINE
30      SUPERSCRIPT                := %08
31      NOT_SUPERSCRIPT           := NOT SUPERSCRIPT
32      SUBSCRIPT                 := %10
33      NOT_SUBSCRIPT             := NOT SUBSCRIPT
34
35      TAB_MASK                   := %80
36      PARITY_MASK                := %7F
37
38       EXTERNAL
39
40      PRINTER_WIDTH             BYTE
41      PRINT_LINE_BUFFER         PROCEDURE
42      FORMFEED                  PROCEDURE
43      LINEFEED                  PROCEDURE
44      MAYBE_ABORT               PROCEDURE
45
46      SOURCE_PTR                ^BYTE
47
48       GLOBAL
49
50
51      CODE                      BYTE
52      BYTES_TAKEN_FROM_SOURCE   WORD

Appendix 6-25

PRINTERR. DRIVER.1
CONSOLE_STATUS_BUFFER ARRAY [ 5 BYTE ]
LINE_BUFFER ARRAY [ 163 BYTE ]
ATTRIBUTE_BUFFER ARRAY [ 163 BYTE ]
COLUMN_NO BYTE
LINE_CONTAINS_PRINTABLE_CHAR BYTE
CHARS_IN_LINE_BUFFER BYTE
ATTRIBUTE BYTE
NEXT_ATTRIBUTE BYTE
ATTRIBUTE_BUFFER_PTR BYTE
LEFTMOST_PRINTABLE_COLUMN BYTE
RIGHTMOST_PRINTABLE_COLUMN BYTE
LINE_BUFFER_PTR BYTE
ATTRIBUTE_SEQUENCE_FLAG BYTE
EOF_FLAG BYTE
LINE_FINISHED_FLAG BYTE

CLEAR_LINE_BUFFER PROCEDURE
ENTRY
LINE_BUFFER_PTR := #LINE_BUFFER [0]
COLUMN_NO := 1
LINE_CONTAINS_PRINTABLE_CHAR := FALSE
LEFTMOST_PRINTABLE_COLUMN := 1
RIGHTMOST_PRINTABLE_COLUMN := 1
DO
IF COLUMN_NO > PRINTER_WIDTH THEN
COLUMN_NO := 1
LINE_BUFFER_PTR := #LINE_BUFFER [0]
ATTRIBUTE_BUFFER_PTR := #ATTRIBUTE_BUFFER [0]
RETURN
FI
LINE_BUFFER_PTR := ( LINE_BUFFER_PTR AND TAB_MASK ) OR ASCII_SPACE
LINE_BUFFER_PTR := INC LINE_BUFFER_PTR
COLUMN_NO += 1
OD
END CLEAR_LINE_BUFFER

PUT_CODE_INTO_LINE_BUFFER PROCEDURE
ENTRY
LINE_BUFFER_PTR := ( LINE_BUFFER_PTR AND TAB_MASK ) OR CODE
ATTRIBUTE_BUFFER_PTR := NEXT_ATTRIBUTE
IF LINE_CONTAINS_PRINTABLE_CHAR = FALSE THEN
LEFTMOST_PRINTABLE_COLUMN := COLUMN_NO
FI

RIGHTMOST_PRINTABLE_COLUMN := COLUMN_NO
COLUMN_NO += 1
LINE_BUFFER_PTR := INC LINE_BUFFER_PTR
ATTRIBUTE_BUFFER_PTR := INC ATTRIBUTE_BUFFER_PTR
END PUT_CODE_INTO_LINE_BUFFER

GOT_TAB PROCEDURE
ENTRY
DO
LINE_BUFFER_PTR := INC LINE_BUFFER_PTR
ATTRIBUTE_BUFFER_PTR := NEXT_ATTRIBUTE
ATTRIBUTE_BUFFER_PTR := INC ATTRIBUTE_BUFFER_PTR
COLUMN_NO += 1
IF (LINE_BUFFER_PTR AND TAB_MASK) <> 0 THEN
RETURN
FI
OD
END GOT_TAB

FETCH_ATTRIBUTE PROCEDURE
ENTRY
ATTRIBUTE_SEQUENCE_FLAG := FALSE
IF CODE
CASE ASCII_CONTROL_R
THEN
IF NEXT_ATTRIBUTE AND BLACK <> 0 THEN
NEXT_ATTRIBUTE := NEXT_ATTRIBUTE AND RED
ELSE
NEXT_ATTRIBUTE := NEXT_ATTRIBUTE OR BLACK
FI
CASE ASCII_CONTROL_B
THEN
IF NEXT_ATTRIBUTE AND BOLD <> 0 THEN
NEXT_ATTRIBUTE := NEXT_ATTRIBUTE AND NOT_BOLD
ELSE
NEXT_ATTRIBUTE := NEXT_ATTRIBUTE OR BOLD
FI
CASE ASCII_CONTROL_U
    THEN
        IF NEXT_ATTRIBUTE AND UNDERLINE <> 0
            THEN
                NEXT_ATTRIBUTE := NEXT_ATTRIBUTE
                                AND NOT_UNDERLINE
            ELSE
                NEXT_ATTRIBUTE := NEXT_ATTRIBUTE
                                OR UNDERLINE
        FI
    CASE ASCII_CONTROL_N
    THEN
        IF NEXT_ATTRIBUTE AND SUBSCRIPT <> 0
            THEN
                NEXT_ATTRIBUTE := NEXT_ATTRIBUTE
                                AND NOT_SUBSCRIPT
            ELSE
                NEXT_ATTRIBUTE := NEXT_ATTRIBUTE
                                OR SUPERSCRIPT
        FI
    CASE ASCII_CONTROL_F
    THEN
        IF NEXT_ATTRIBUTE AND SUPERSCRIPT <> 0
            THEN
                NEXT_ATTRIBUTE := NEXT_ATTRIBUTE
                                AND NOT_SUPERSCRIPT
            ELSE
                NEXT_ATTRIBUTE := NEXT_ATTRIBUTE
                                OR SUBSCRIPT
        FI
    END FETCH_ATTRIBUTE

GET_CODE PROCEDURE
ENTRY
    CODE := SOURCE_PTR^ 
    IF CODE = %FF
        THEN
            EOF_FLAG := TRUE
            RETURN
    FI
    CODE := CODE AND PARITY_MASK
    SOURCE_PTR := INC SOURCE_PTR
    BYTES_TAKEN_FROM_SOURCE += 1
    IF ATTRIBUTE_SEQUENCE_FLAG = TRUE
THEN
    FETCH_ATTRIBUTE
    RETURN
FI
IF CODE > ASCII_SPACE
THEN
    PUT_CODE_INTO_LINE_BUFFER
    RETURN
FI
IF CODE
CASE ASCII_SPACE
THEN
    LINE_BUFFER_PTR := INC LINE_BUFFER_PTR
    ATTRIBUTE_BUFFER_PTR := NEXTATTRIBUTE
    ATTRIBUTE_BUFFER_PTR := INC ATTRIBUTE_BUFFER_PTR
    COLUMN_NO += 1
CASE ASCII_CR
THEN
    PRINT_LINE_BUFFER
    LNEFEED
    MAYBE_ABORT
CASE ASCII_FF
THEN
    PRINT_LINE_BUFFER
    FORMFEED
    MAYBE_ABORT
CASE ASCII_LF
THEN
    PRINT_LINE_BUFFER
    LINEFEED
    MAYBE_ABORT
CASE ASCII_TAB
THEN
    GOT_TAB
CASE ASCII_ESC
THEN
    ATTRIBUTE_SEQUENCE_FLAG := TRUE
FI
END GET_CODE
END PRINTER_DRIVER_1

Appendix

0 ERROR(S)
0 WARNING(S)

6-29

PRINTER.DRIVER.1
PLZSYS 2.02

1  PRINTED_DRIVER_2    MODULE

3  CONSTANT

5  ASCII_SPACE := ' '  

7  PARITY_MASK := $7F

8  FORWARD := 1

9  BACKWARD := 0

11  TRUE := 1

12  FALSE := 0

14  WRITE_LINE := $10

16  EXTERNAL

18  LEFTMOST_PRINTABLE_COLUMN BYTE

19  RIGHTMOST_PRINTABLE_COLUMN BYTE

20  PRESENT_COLUMN BYTE

21  DIRECTION BYTE

22  CHARS_IN_LINE_BUFFER BYTE

23  LINE_BUFFER_PTR '^BYTE

24  ATTRIBUTE_BUFFER_PTR '^BYTE

25  ATTRIBUTE BYTE

26  REQUEST_CODE BYTE

27  LINE_CONTAINS_PRINTABLE_CHAR BYTE

29  LINE_BUFFER ARRAY [ 163 BYTE ]

31  ATTRIBUTE_BUFFER ARRAY [ 163 BYTE ]

35  SEND PROCEDURE ( BYTE )

36  PRINT PROCEDURE ( BYTE BYTE )

37  \t! CHAR, ATTRIBUTE !

39  REQUEST_FORWARD PROCEDURE

40  REQUEST_BACKWARD PROCEDURE

41  WAIT_FOR_ACK PROCEDURE

42  CLEAR_LINE_BUFFER PROCEDURE

44  INTERNAL

47  CHAR BYTE

48  COLUMN_NO BYTE

49  NEXT_COLUMN_NO BYTE

50  SPACE_COUNT BYTE

51  SPACE_SKIP_FLAG BYTE
PROCEDURE
ENTRY

IF LEFTMOST_PRINTABLE_COLUMN > PRESENT_COLUMN
    THEN
        ABSOLUTE_TAB ( LEFTMOST_PRINTABLE_COLUMN )
        REQUEST_FORWARD
        RETURN
    FI

IF PRESENT_COLUMN > RIGHTMOST_PRINTABLE_COLUMN
    THEN
        ABSOLUTE_TAB ( RIGHTMOST_PRINTABLE_COLUMN )
        REQUEST_BACKWARD
        RETURN
    FI

IF (RIGHTMOST_PRINTABLE_COLUMN - PRESENT_COLUMN )
    > ( PRESENT_COLUMN - LEFTMOST_PRINTABLE_COLUMN )
    THEN
        ABSOLUTE_TAB ( LEFTMOST_PRINTABLE_COLUMN )
        REQUEST_FORWARD
        RETURN
    FI

ABSOLUTE_TAB ( RIGHTMOST_PRINTABLE_COLUMN )
REQUEST_BACKWARD

END SET_UP_DIRECTION

GLOBAL

PRINT_LINE_BUFFER PROCEDURE
ENTRY

 IF LINE_CONTAINS_PRINTABLE_CHAR = FALSE
    THEN
        CLEAR_LINE_BUFFER RETURN
    ELSE
        CHARS_IN_LINE_BUFFER := RIGHTMOST_PRINTABLE_COLUMN
        - LEFTMOST_PRINTABLE_COLUMN + 1
    FI

SET_UP_DIRECTION
LINE_BUFFER_PTR := #LINE_BUFFER [ PRESENT_COLUMN-1 ]
ATTRIBUTE_BUFFER_PTR := #ATTRIBUTE_BUFFER [ PRESENT_COLUMN-1 ]
NEXT_COLUMN_NO := PRESENT_COLUMN
SPACE_SKIP_FLAG := FALSE
SPACE_COUNT := 0
DO
  IF CHARS_IN_LINE_BUFFER = 0 THEN
    CLEAR_LINE_BUFFER
    RETURN
  FI

  COLUMN_NO := NEXT_COLUMN_NO
  CHAR := LINE_BUFFER_PTR \& PARITY_MASK
  ATTRIBUTE := ATTRIBUTE_BUFFER_PTR
  IF DIRECTION = BACKWARD THEN
    LINE_BUFFER_PTR := DEC LINE_BUFFER_PTR
    ATTRIBUTE_BUFFER_PTR := DEC ATTRIBUTE_BUFFER_PTR
    NEXT_COLUMN_NO -= 1
  ELSE
    LINE_BUFFER_PTR := INC LINE_BUFFER_PTR
    ATTRIBUTE_BUFFER_PTR := INC ATTRIBUTE_BUFFER_PTR
    NEXT_COLUMN_NO += 1
  FI

  CHARS_IN_LINE_BUFFER -= 1

  IF CHAR = ASCII_SPACE THEN
    IF SPACE_SKIP_FLAG = FALSE THEN
      SPACE_SKIP_FLAG := TRUE
      SPACE_COUNT := 1
      REPEAT
    FI

    SPACE_COUNT += 1
    REPEAT
  FI

  IF SPACE_SKIP_FLAG = TRUE THEN
    SPACE_SKIP_FLAG := FALSE
    IF SPACE_COUNT >= 3 THEN
      ABSOLUTE_TAB ( COLUMN_NO )
    ELSE
      DO
        IF SPACE_COUNT = 0 THEN EXIT FI
        PRINT ( ASCII_SPACE ATTRIBUTE )
      OD
      SPACE_COUNT -= 1
    FI
FI
PRINT ( CHAR_ATTRIBUTE )
OD
END PRINT_LINE_BUFFER
END PRINTER_DRIVER_2

END OF ZCODE GENERATION
0 ERROR(S) 0 WARNING(S)
PLZSYS 2.02

1  SPINWRITER  MODULE
2
3  !  Extended 3/3/79 to allow for subscripting and superscripting!
4  !  Also for page-waits and controllable auto-formfeed!
5
6
7  CONSTANT
8
9
10    PRINTER_BUFFER_SIZE  :=  256
11    PITCH  :=  12
12    SS  :=  120/PITCH
13
14    ASCII_ETX  :=  %03
15    ASCII_ACK  :=  %06
16    ASCII_ESCAPE  :=  %1B
17    ASCII_FF  :=  %0C
18    ASCII_BS  :=  %08
19    ASCII_UL  :=  ' '.
20    ASCII_SPACE  :=  ' '
21
22    PARITY_MASK  :=  %7F
23
24    FORWARD  :=  1
25    BACKWARD  :=  0
26
27    BLACK  :=  %01
28    BOLD  :=  %02
29    UNDERLINE  :=  %04
30    SUPERSCRIPT  :=  %08
31    NOT_SUPERSCRIPT  :=  NOT SUPERSCRIPT
32    SUBSCRIPT  :=  %10
33    NOT_SUBSCRIPT  :=  NOT SUBSCRIPT
34
35    TRUE  :=  1
36    FALSE  :=  0
37
38  INTERNAL
39
40
41    LINE_COUNT  BYTE
42    COLOUR  BYTE
43    HMI  BYTE
44    SUBSCRIPT_FLAG  BYTE
45    SUPERSCRIPT_FLAG  BYTE
46
47    LAST_SCRIPT_STATE  BYTE
48
49  EXTERNAL
50
51
52    OUTCH2  PROCEDURE ( BYTE )
INCH2 PROCEDURE RETURNS (BYTE)

PAGE_WAIT PROCEDURE

NEXT_ATTRIBUTE BYTE

PAGE_WAIT_FLAG BYTE

AUTO_FF_FLAG BYTE

GLOBAL

PRINTER_WIDTH BYTE := 163

DIRECTION BYTE

PRESENT_COLUMN BYTE

AUTO_FF_LINE_COUNT BYTE := 63

BYTE_COUNT BYTE

SEND_ETX PROCEDURE

ENTRY

OUTCH2 (ASCII_ETX)

BYTE_COUNT := 0

END SEND_ETX

WAIT_FOR_ACK PROCEDURE

ENTRY

DO

IF (INCH2 AND PARITY_MASK) = ASCII_ACK

THEN

RETURN

FI

OD

END WAIT_FOR_ACK

SEND PROCEDURE (CODE BYTE)

ENTRY

IF BYTE_COUNT > PRINTER_BUFFER_SIZE - 10

THEN

SEND_ETX

WAIT_FOR_ACK

FI

BYTE_COUNT += 1

OUTCH2 (CODE)

END SEND

REQUEST_HALF_LINEFEED_PITCH PROCEDURE

ENTRY

SEND (ASCII_ESC)
106 SEND ( ']' )
107 SEND ( 'R' )
108 END REQUEST_HALF_LINEFEED_PITCH
109
110 REQUEST_STANDARD_LINEFEED_PITCH PROCEDURE
111 ENTRY
112 SEND ( ASCII_ESC )
113 SEND ( ']' )
114 SEND ( 'w' )
115 END REQUEST_STANDARD_LINEFEED_PITCH
116
117 REQUEST_POS_HALF_LINE PROCEDURE
118 ENTRY
119 REQUEST_HALF_LINEFEED_PITCH
120 SEND ( '%L' )
121 REQUEST_STANDARD_LINEFEED_PITCH
122 END REQUEST_POS_HALF_LINE
123
124 REQUEST_NEG_HALF_LINE PROCEDURE
125 ENTRY
126 REQUEST_HALF_LINEFEED_PITCH
127 SEND ( ASCII_ESC )
128 SEND ( 'g' )
129 REQUEST_STANDARD_LINEFEED_PITCH
130 END REQUEST_NEG_HALF_LINE
131
132 FORMFEED PROCEDURE
133 ENTRY
134 SEND ( ASCII_FF )
135 LINE_COUNT := 0
136 LAST_SCRIPT_STATE := 0
137 SUPERSCRIPT_FLAG := FALSE
138 SUBSCRIPT_FLAG := FALSE
139 IF PAGE_WAIT_FLAG = TRUE
140 THEN
141 PAGE_WAIT
142 FI
143 END FORMFEED
144
145 LINEFEED PROCEDURE
146 ENTRY
147 IF AUTO_FF_FLAG = TRUE
148 THEN
149 IF LINE_COUNT >= AUTO_FF_LINE_COUNT
150 THEN
151 FORMFEED
152 FI
153 FI
154 IF AUTO_FF_FLAG = TRUE
155 THEN
156 IF LINE_COUNT >= AUTO_FF_LINE_COUNT
157 THEN
158 FORMFEED
RETURN
FI
FI
SEND ( '%L' )
LINE_COUNT += 1
END LINEFEED

REQUEST_FORWARD PROCEEDURE
ENTRY
IF DIRECTION = FORWARD THEN RETURN FI
SEND ( ASCII_ESC )
SEND ( '>' )
DIRECTION := FORWARD
END REQUEST_FORWARD

REQUEST_BACKWARD PROCEEDURE
ENTRY
IF DIRECTION = BACKWARD THEN RETURN FI
SEND ( ASCII_ESC )
SEND ( '<' )
DIRECTION := BACKWARD
END REQUEST_BACKWARD

REQUEST_BLACK PROCEEDURE
ENTRY
SEND ( ASCII_ESC )
SEND ( '4' )
COLOUR := BLACK
END REQUEST_BLACK

REQUEST_RED PROCEEDURE
ENTRY
SEND ( ASCII_ESC )
SEND ( '3' )
COLOUR := 0
END REQUEST_RED

DEFINE_HMI PROCEEDURE ( SPACING BYTE )
ENTRY
SEND ( ASCII_ESC )
SEND ( '1' )
SEND ( SPACING + '%40' )
END DEFINE_HMI

SEND_BOLD_CHAR PROCEEDURE ( CHAR BYTE ATTRIBUTE BYTE )
ENTRY
DEFINE_HMI ( 0 )
SEND ( CHAR )
IF ( ATTRIBUTE AND UNDERLINE ) <> 0 THEN
    SEND ( ASCII_UL )
FI
DEFINE_HMI ( 1 )
SEND ( CHAR )
DEFINE_HMI ( SS-1 )
SEND ( CHAR )
DEFINE_HMI ( SS )

END SEND_BOLD_CHAR

ABSOLUTE_TAB
ENTRY
IF COLUMN > 162 THEN COLUMN := 162 FI
SEND ( ASCII_ESC )
PRESENT_COLUMN := COLUMN
IF COLUMN < 33 THEN
    SEND ( 'P' )
    SEND ( %3F + COLUMN )
    RETURN
FI
IF COLUMN < 65 THEN
    SEND ( 'Q' )
    SEND ( %1F + COLUMN )
    RETURN
FI
IF COLUMN < 97 THEN
    SEND ( 'R' )
    SEND ( COLUMN - 1 )
    RETURN
FI
IF COLUMN < 129 THEN
    SEND ( 'S' )
    SEND ( COLUMN - %21 )
    RETURN
FI
IF COLUMN < 161 THEN

SEND ( 'T' )
SEND ( COLUMN - $41 )
RETURN
FI
SEND ( 'U' )
SEND ( COLUMN - $61 )
END ABSOLUTE_TAB
PRINT PROCEDURE ( CHAR BYTE ATTRIBUTE BYTE )
ENTRY
DO
IF ATTRIBUTE AND (SUBSCRIPT OR SUPERSCRIPT) = LAST_SCRIPT_STATE THEN
EXIT
FI
IF ATTRIBUTE AND SUPERSCRIPT <> 0 THEN
IF SUBSCRIPT_FLAG = TRUE THEN
REQUEST_NEG_HALF_LINE
SUBSCRIPT_FLAG := FALSE
FI
REQUEST_NEG_HALF_LINE
SUPERSCRIPT_FLAG := TRUE
EXIT
FI
IF ATTRIBUTE AND SUBSCRIPT <> 0 THEN
IF SUPERSCRIPT_FLAG = TRUE THEN
REQUEST_POS_HALF_LINE
SUPERSCRIPT_FLAG := FALSE
FI
REQUEST_POS_HALF_LINE
SUBSCRIPT_FLAG := TRUE
EXIT
FI
IF SUPERSCRIPT_FLAG = TRUE THEN
REQUEST_POS_HALF_LINE
SUPERSCRIPT_FLAG := FALSE
EXIT
FI
REQUEST_NEG_HALF_LINE
318     SUBSCRIPT_FLAG := FALSE
319
320     EXIT
321
322     OD
323
324     LAST_SCRIPT_STATE := ATTRIBUTE
325     AND ( SUPERSCRIPT OR SUBSCRIPT )
326
327     IF ( ATTRIBUTE AND BLACK ) <> COLOUR
328     THEN
329         IF ( ATTRIBUTE AND BLACK ) = BLACK
330         THEN
331             REQUEST_BLACK
332         ELSE
333             REQUEST_RED
334         FI
335     FI
336
337     IF CHAR > ASCII_SPACE
338     THEN
339         IF ATTRIBUTE AND BOLD <> 0
340         THEN
341             SEND_BOLD_CHAR ( CHAR ATTRIBUTE )
342         ELSE
343             IF ( ATTRIBUTE AND UNDERLINE ) <> 0
344         THEN
345             DEFINE_HMI ( 0 )
346             SEND ( CHAR )
347             DEFINE_HMI ( SS )
348             SEND ( ASCII_UL )
349         ELSE
350             SEND ( CHAR )
351         FI
352     FI
353     ELSE
354     SEND ( CHAR ) ! SPACES WILL NOT BE UNDERLINED!
355     FI
356
357     IF DIRECTION = BACKWARD
358     THEN
359         PRESENT_COLUMN -= 1
360     ELSE
361         PRESENT_COLUMN += 1
362     FI
363
364     IF PRESENT_COLUMN = 0 THEN PRESENT_COLUMN += 1 FI
365
366     END PRINT
367
368
369
370
Spinwriter/Diablo driver

371 END SPINWRITER
END OF ZCODE GENERATION
  0 ERROR(S)       0 WARNING(S)
PLZSYS 2.02

DIABLO MODULE

! Extended 3/3/79 to allow for subscripting and superscripting!
! Also for operator controlled page_waits, and auto formfeeds.

CONSTANT

PRINTER_BUFFER_SIZE := 158
PITCH := 12
SS := 120/PITCH

ASCII_ETX := $03
ASCII_ACK := $06
ASCII_ESC := $1B
ASCII_FF := $0C
ASCII_BS := $08
ASCII_UL := '\_'
ASCII_SPACE := '\ ';
ASCII_US := $1F
ASCII_TAB := $09

PARITY_MASK := $7F

FORWARD := 1
BACKWARD := 0

BLACK := $01
BOLD := $02
UNDERLINE := $04

SUPERScript := $08
NOT_SUPERScript := NOT SUPERScript
SUBSCRIPT := $10
NOT_SUBSCRIPT := NOT SUBSCRIPT

TRUE := 1
FALSE := 0

INTERNAL

LINE_COUNT BYTE
COLOUR BYTE
HMI BYTE

SUPERScript_FLAG BYTE
SUBSCRIPT_FLAG BYTE

LAST_SCRIPT_STATE BYTE

EXTERNAL
OUTCH2
INCH2
PAGE_WAIT
PAGE_WAIT_FLAG
AUTO_FF_FLAG
GLOBAL
PRINTER_WIDTH
DIRECTION
PRESENT_COLUMN
AUTO_FF_LINE_COUNT
BYTE_COUNT
SEND_ETX
PROCEDURE
ENTRY
OUTCH2 ( ASCII_ETX )
BYTE_COUNT := 0
END SEND_ETX

WAIT_FOR_ACK
PROCEDURE
ENTRY
DO
IF ( INCH2 AND PARITY_MASK ) = ASCII_ACK
THEN
RETURN
FI
OD
END WAIT_FOR_ACK

SYNCH
PROCEDURE
ENTRY
IF BYTE_COUNT < PRINTER_BUFFER_SIZE - 10 THEN RETURN FI
SEND_ETX
WAIT_FOR_ACK
END SYNCH

SEND
PROCEDURE ( CODE BYTE )
ENTRY
IF BYTE_COUNT > PRINTER_BUFFER_SIZE - 10
THEN
SEND_ETX
WAIT_FOR_ACK
FI
BYTE_COUNT += 1
OUTCH2 ( CODE )
END SEND

FORMFEED PROCEDURE
ENTRY
SEND ( ASCII_FF )
LINE_COUNT := 0
LAST_SCRIPT_STATE := 0
SUPERSCRIPT_FLAG := FALSE
SUBSCRIPT_FLAG := FALSE

IF PAGE_WAIT_FLAG = TRUE THEN PAGE_WAIT FI
END FORMFEED

REQUEST_FORWARD PROCEDURE
ENTRY
IF DIRECTION = FORWARD THEN RETURN FI
SYNCH
OUTCH2 ( ASCII_ESC )
OUTCH2 ( '5' )
BYTE_COUNT += 2
DIRECTION := FORWARD
END REQUEST_FORWARD

REQUEST_BACKWARD PROCEDURE
ENTRY
IF DIRECTION = BACKWARD THEN RETURN FI
SYNCH
OUTCH2 ( ASCII_ESC )
OUTCH2 ( '6' )
BYTE_COUNT += 2
DIRECTION := BACKWARD
END REQUEST_BACKWARD

REQUEST_BLACK PROCEDURE
ENTRY
SYNCH
OUTCH2 ( ASCII_ESC )
OUTCH2 ( 'B' )
BYTE_COUNT += 2
COLOUR := BLACK
END REQUEST_BLACK

REQUEST_RED PROCEDURE
ENTRY
SYNCH
OUTCH2 ( ASCII_ESC )
159   OUTCH2 ( 'A' )
160   BYTE_COUNT += 2
161   COLOUR := 0
162   END REQUEST_RED
163
164   LINEFEED PROCEDURE
165      ENTRY
166      IF AUTO_FF_FLAG = TRUE
167      THEN
168          IF LINE_COUNT >= AUTO_FF_LINE_COUNT
169              THEN
170                  FORMFEED
171                  RETURN
172          FI
173      FI
174      SEND ( '%L' )
175      LINE_COUNT += 1
176   END LINEFEED
177
178   REQUEST_POS_HALF_LINE PROCEDURE
179      ENTRY
180      SYNCH
181      OUTCH2 ( ASCII_ESC )
182      OUTCH2 ( 'U' )
183      BYTE_COUNT += 2
184   END REQUEST_POS_HALF_LINE
185
186   REQUEST_NEG_HALF_LINE PROCEDURE
187      ENTRY
188      SYNCH
189      OUTCH2 ( ASCII_ESC )
190      OUTCH2 ( 'D' )
191      BYTE_COUNT += 2
192   END REQUEST_NEG_HALF_LINE
193
194   DEFINE_HMI PROCEDURE ( SPACING BYTE )
195      ENTRY
196      SYNCH
197      OUTCH2 ( ASCII_ESC )
198      OUTCH2 ( ASCII_US )
199      OUTCH2 ( SPACING + 1 )
200      BYTE_COUNT += 3
201   END DEFINE_HMI
202
203   SEND_BOLD_CHAR PROCEDURE ( CHAR BYTE ATTRIBUTE BYTE )
204      ENTRY
205      DEFINE_HMI ( 0 )
206
SEND ( CHAR )
IF ( ATTRIBUTE AND UNDERLINE ) <> 0
THEN
    SEND ( ASCII_UL )
FI
DEFINE_HMI ( 1 )
SEND ( CHAR )
DEFINE_HMI ( SS-1 )
SEND ( CHAR )
DEFINE_HMI ( SS )
END SEND_BOLD_CHAR

ABSOLUTE_TAB

LOCAL RESIDUE BYTE
    DIR BYTE
ENTRY
    IF COLUMN > 156 THEN COLUMN := 156 FI
    SYNCH
    OUTCH2 ( ASCII_ESC )
    OUTCH2 ( ASCII_TAB )
    BYTE_COUNT += 2
PRESENT_COLUMN := COLUMN
IF COLUMN <= 126
    THEN
        OUTCH2 ( COLUMN )
        BYTE_COUNT += 1
        RETURN
    FI
RESIDUE := COLUMN - 126
OUTCH2 ( 126 )
BYTE_COUNT += 1
DIR := DIRECTION
REQUEST_FORWARD
DO
    IF RESIDUE = 0
    THEN
        IF DIR = BACKWARD
        THEN
            REQUEST_BACKWARD
        FI
    FI
    RETURN
    IF
SEND ( ASCII_SPACE )
RESIDUE := 1
OD
END ABSOLUTE_TAB

PRINT PROCEDURE ( CHAR BYTE ATTRIBUTE BYTE )
ENTRY
DO
IF ATTRIBUTE AND (SUBSCRIPT OR SUPERScript) = LAST_SCRIPT_STATE
    THEN
        EXIT
    FI

IF ATTRIBUTE AND SUPERScript <> 0
    THEN
        IF SUBSCRIPT_FLAG = TRUE
            THEN
                REQUEST_NEG_HALF_LINE
                SUBSCRIPT_FLAG := FALSE
            FI
        REQUEST_NEG_HALF_LINE
        SUPERScript_FLAG := TRUE
        EXIT
    FI

IF ATTRIBUTE AND SUBSCRIPT <> 0
    THEN
        IF SUPERSCRIPT_FLAG = TRUE
            THEN
                REQUEST_POS_HALF_LINE
                SUPERSCRIPT_FLAG := FALSE
            FI
        REQUEST_POS_HALF_LINE
        SUBSCRIPT_FLAG := TRUE
        EXIT
    FI

IF SUPERSCRIPT_FLAG = TRUE
    THEN
        REQUEST_POS_HALF_LINE
        SUPERSCRIPT_FLAG := FALSE
    EXIT

REQUEST_NEG_HALF_LINE
SUBSCRIPT_FLAG := FALSE
EXIT
OD
LAST_SCRIPT_STATE
:= ATTRIBUTE AND ( SUPERSCRPT OR SUBSCRIPT )
IF ( ATTRIBUTE AND BLACK ) <> COLOUR
THEN
  IF ( ATTRIBUTE AND BLACK ) = BLACK
  THEN
    REQUEST_BLACK
  ELSE
    REQUEST_RED
  FI
FI
IF CHAR > ASCII_SPACE
THEN
  IF ATTRIBUTE AND BOLD <> 0
  THEN
    SEND_BOLD_CHAR ( CHAR ATTRIBUTE )
  ELSE
    IF ( ATTRIBUTE AND UNDERLINE ) <> 0
    THEN
      DEFINE_HMI ( 0 )
      SEND ( CHAR )
      DEFINE_HMI ( SS )
      SEND ( ASCII_UL )
    ELSE
      SEND ( CHAR )
    FI
  FI
ELSE
  SEND ( CHAR ) ! SPACES WILL NOT BE UNDERLINED !
FI
IF DIRECTION = BACKWARD
THEN
  PRESENT_COLUMN -= 1
ELSE
  PRESENT_COLUMN += 1
FI
IF PRESENT_COLUMN = 0 THEN PRESENT_COLUMN += 1 FI
END PRINT
END DIABLO
END OF ZCODE GENERATION
0 ERROR(S) 0 WARNING(S)
Call to SYSTEM

Loc  Obj Code M Stmt Source Statement

1  *H Call to SYSTEM
2  
3  *I PLZ INTERFACE MACROS
133  *LIST ON
134  *MACLIST OFF
135  ; Date code:- October 22nd. 1978.
136  
137  ; This interface module allows a PLZ programme to make
138  ; calls to RIO.
139  
140  ; Declare CALRIO PROCEDURE ( VECTOR_PTR "byte )
141  ; RETURNS ( COMPLETION_CODE byte )
142  
143  ; There must be a standard RIO request vector stored
144  ; starting at the location beginning VECTOR_PTR'
145  
146  global CALRIO calrio
147  
148  SYSTEM EQU 1403H
149  
150  
151  CALRIO
152  calrio
0000  
153  ENT 0 ; no locals
154  0008  000E  000F  0011  0013  0016  0018  001B  001E
155            156            157            159            160            161            163            164            166
LDHL 4  push hl  pop iy ; and then where it should be
158  
159        push ix ; save it
160  
160        call SYSTEM ; go and do the necessary
161        pop ix ; restore it
162  
163        ld a,(iy+10) ; get the completion code
164        STA 6 ; and place it as return parameter
165  
166        RTN 0 2 ; return to caller. 0 locals, 2 I/P param bytes
167  
168  
169        END
This interface module receives I/O calls from RIO, and passes the IY register value to the called programme as a single parameter.

The intention of the module is to act as an interface to enable I/O drivers to be written largely in PLZ.

```
*H RIO.IO INTERFACE

; Date_code: - October 31st. 1978

; This interface module receives I/O calls from RIO, and
; passes the IY register value to the called programme
; as a single parameter.

; The intention of the module is to act as an interface
; to enable I/O drivers to be written largely in PLZ.

EXTERNAL PLZDVR
GLOBAL ENTRY entry
ENTRY entry

push iy ; the actual parameter
LD (IY_SAV),IY ; save iy
call PLZDVR ; pass control to the driver proper
LD IY,(IY_SAV) ; restore iy
bit 0,(iy+1) ; was it int.req ?
ld a,(iy+10) ; get c_code
jr nz,intreq
cp 80h
ret z ; if so, go back quietly
getera
ld h,(iy+9)
ld l,(iy+8)
jmpret
ld a,h
or l
ret z ; rtn add field was zero
pop bc ; balance stack
jp (hl)
intreq cp 80h
jr nz, getera
ld h,(iy+7)
jr jmpret ; check cra field
ld l,(iy+6)
ld h,(iy+9)
jr getera
ld l,(iy+8)
ld h,(iy+10)
```

Appendix 6-50 RIO.IO INTERFACE
Spinwriter/Diablo driver

Revision 2.1

0030
51  defs 2
52
53  end
The calling programme should contain the following declarations:

```asm
1 "H SIB CH2 input/output
2 *P 50
3
4 "I PLZ.INTERFACE.MACROS
134 LIST ON
135 MACLIST OFF
136
137 ; Date_code:- October 23rd. 1978.
138
139 ; This module contains the routines which enable a PLZ
140 ; programme to be involved with I/O through channel 2
141 ; of an SIB installed in the system.
142
143 ;
144 ;
145
146 ; EXTERNAL
147
148 ; OUTCH2 PROCEDURE ( BYTE )
149 ; ! send, with wait ready!
150
151 ; INCH2 PROCEDURE RETURNS ( BYTE )
152 ; ! input, with wait ready!
153
154 ; INCH2E PROCEDURE RETURNS ( BYTE )
155 ; ! input, wait ready, echo!
156
157 ; STACH2 PROCEDURE RETURNS ( BYTE )
158 ; ! read status of USART2!
159
160 ; SNDCH2 PROCEDURE ( BYTE )
161 ; ! send, without wait ready!
162
163 ; RDCH2 PROCEDURE RETURNS ( BYTE )
164 ; ! read, without wait ready!
165
166 ; SETCH2 PROCEDURE
167 ; ! set up USART2 + CTC baud rate!
168
169
170 global OUTCH2 INCH2 INCH2E
171 global STACH2 SNDCH2 RDCH2 SETCH2
172 global outch2 inch2 inch2e
173 global stach2 sndch2 rdch2 setch2
```
Spinwriter/Diablo driver

SIB CH2 input/output

LOC OBJ CODE M STMT SOURCE STATEMENT

SIB

PAGE 2

ASM 5.8

174  *E
175
176  SETCH2
177  setch2
0000
178  ENT 0
179
0008  CD9400  R 180  call BAUDR  ; set up the baud rate
000B  CD7B00  R 181  call SUART2  ; set up USART-2
182
000E
183  RTN 0 0  ; no locals, no IN parameters
184
185
186  OUTCH2
187  outch2
0011
188  ENT 0
189
0019
190  LDA 4  ; get the code for issuing
001C  CDA600  R 191  call OUSIB2  ; send it with wait ready
192
001F
193  RTN 0 2  ; no locals, 2 bytes IN
194
195
196  INCH2
197  inch2
0024
198  ENT 0
199
002C  CD9D00  R 200  call INSIB2  ; get the code, with wait ready
002F
201  STA 4  ; place it as return parameter
202
0032
203  RTN 0 0  ; no locals, no IN parameters
204
205
206  INCH2E
207  inch2e
0035
208  ENT 0
209
003D  CD9D00  R 210  call INSIB2  ; get the code, with wait ready
0040  CDA600  R 211  call OUSIB2  ; echo it, with wait ready
0043
212  STA 4  ; place code as return parameter
213
0046
214  RTN 0 0  ; no locals, no IN parameters
215
216
217  STACH2
218  stach2
0049
219  ENT 0
220
0051  DB91
221  in a,(USART2+1)  ; get the status reg contents
0053
222  STA 4  ; place it as return parameter
223

Appendix 6-53
Spinwriter/Diablo driver

SIB CH2 input/output

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ CODE</th>
<th>M</th>
<th>STMT</th>
<th>SOURCE</th>
<th>STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0056</td>
<td>224</td>
<td>0</td>
<td>0</td>
<td></td>
<td>RTN</td>
</tr>
<tr>
<td></td>
<td>225</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>226</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>227</td>
<td></td>
<td>SNDCH2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>228</td>
<td></td>
<td>sndch2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0059</td>
<td>229</td>
<td></td>
<td>ENT 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>230</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0061</td>
<td>231</td>
<td></td>
<td>LDA 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0064</td>
<td>D390</td>
<td></td>
<td>out (USART2),a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0066</td>
<td>234</td>
<td></td>
<td>RTN 0 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>235</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>236</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>237</td>
<td></td>
<td>RDCH2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>238</td>
<td></td>
<td>rdch2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0068</td>
<td>239</td>
<td></td>
<td>ENT 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>240</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0073</td>
<td>DB90</td>
<td></td>
<td>in a,(USART2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0075</td>
<td></td>
<td></td>
<td>STA 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0078</td>
<td></td>
<td></td>
<td>RTN 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>245</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>246</td>
<td></td>
<td>&quot;I SIB.CONTROL&quot;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

; no locals, no IN parameters

; get the code to be sent

; send it immediately

; no locals, 2 bytes IN parameters

; get data reg contents

; place it as return parameter

; no locals, no IN parameters
247  *H SIB CONTROL
248
250
251 ;*****************************************************************************
252 ;  RESET AND SET UP USART2
253 ;*****************************************************************************
254
255 SUART2  LD C,USART2+1 ; USART2 - CONTROL REGISTER
256 007B 0B91
257 007D AF
258 007E ED79
259 0080 ED79
260 0084 3E40
261 OUT (C),A ; THREE INTERNAL RESETS = EXT RESET
262 0086 ED79
263 0088 3ECE
264 008A ED79
265 OUT (C),A ; INT RESET
266 008C 3E37
267 OUT (C),A ; ENTER MODE INSTRUCTION FORMAT
268 0090 ED79
269 DEC C
270 0091 ED78
271 IN A,(C) ; CLEAR OUT GARBAGE CHARACTER
272 0093 C9
273
274 USART0  EQU 8CH
275 USART1  EQU 8EH
276 USART2  EQU 90H
277 USART3  EQU 92H
278
279
280
281
282 ;*****************************************************************************
283 ;  SET UP CTC1 TO GENERATE THE BAUD RATE
284 ;*****************************************************************************
285
286 BAUDR  LD A,TIMMOD ; TIMER MODE ETC
287 0094 3E07
288 0096 D381
289 0098 3E04
290 009A D381
291 009C C9
292 BOUT (CTC1),A ; WITH RATE IN RATEO
293 294 CTC0  EQU 80H ; ADDRESS OF CTC0
295 296 CTC2  EQU 82H ; ADDRESS OF CTC2
297 298 CTC1  EQU 81H ; ADDRESS OF CTC1
299 296 CTC3  EQU 83H ; ADDRESS OF CTC3
300
301 Appendix 6-55
TIMMOD EQU 07H  ; TIMER MODE, PRESCALER=16
RATEO EQU 4     ; FOR 1200 BAUDS

; ELEMENTARY CHARACTER LEVEL I/O

INSIB2 IN A,(USART2+1) ; GET STATUS REGISTER
BIT RXRDY,A ; IS THE RECEIVER FLAG SET
JR Z,INSIB2 ; IF NOT, WAIT FOR IT
IN A,(USART2) ; GET CONTENT OF DATA REGISTER
RET

OUSIB2 PUSH AF ; SAVE CHARACTER
BIT TXRDY,A ; TEST THE TRANSMITTER READY FLAG
JR Z,BZY ; IF UNREADY THEN WAIT
POP AF ; RESTORE CHARACTER CODE
OUT (USART2),A ; SEND IT
RET

RXRDY EQU 1 ; RECEIVER READY BIT
TXRDY EQU 0 ; TRANSMITTER READY BIT
MST macro \$n

; \$n is in BYTES

cond \$n=2).or.(\$n=4)
push hl
cond \$n=4
push hl
endc
cond .not.(\$n=0).and..not.(\$n=2).and..not.(\$n=4)
ld hl,-\$n
add hl,sp
ld sp,hl
endc
endm

Mark-stack macro:
Allocate room on stack for out parameters
before a procedure call.

; Optimise the code when 0,1, or 2 parameters
; i.e. 0,2 or 4 bytes.
*E

; Procedure entry:

; Allocate locals on stack (No. of bytes)

; Optimise when 0, 2, or 4 bytes.

ENTRY macro #n ; #n is in BYTES ***

push ix
ld ix, 0
add ix, sp
cond (#n=2).or. (#n=4)
push hl
cond #n=4
push hl
endc
cond .not. (#n=0).and..not. (#n=2).and..not. (#n=4)
1d hl, -#n
add hl, sp
1d sp, hl
endc
endm
Procedure return:

Deallocate locals (bytes) and IN parameters.

Optimise when 0, 2, or 4 bytes.

RTN macro  

; #L, #n are in BYTES ***

cond #L
ld sp,ix
endc

pop ix

cond #n=0
ret
endc

cond (#n=2).or. (#n=4)
pop hl
pop de

cond #n=4
pop de
endc

cond (#n=2).or. (#n=4)
jp (hl)
endc

cond .not. (#n=0).and..not. (#n=2).and..not. (#n=4)
pop de
ld hl,#n
add hl,sp
ld sp,hl
ex de,hl
jp (hl)
endc
endm
Macros for accessing locals and parameters from the stack. This is only a small selection.

; Load hl from #n (offset of word variable from ix)
LDHL macro #n
  ld l,(ix+#n)
  ld h,(ix+#n+1)
endm

; Store hl into #n (offset of word variable from ix)
STHL macro #n
  ld (ix+#n),l
  ld (ix+#n+1),h
endm

; Load A from #n (offset of byte variable from ix)
LDA macro #n
  ld a,(ix+#n)
endm

; Store A into #n (offset of byte variable from ix)
STA macro #n
  ld (ix+#n),a
endm

*LIST ON
Zilog's Real Time Software (ZRTS) provides a set of modular software components that allows quick and easy implementation of customized operating systems for all members of the Z8000 16-bit microprocessor family. In effect, ZRTS extends the instruction set of the Z8000, adding easy-to-use commands that give the Z8000 the capability for managing real-time, multi-tasking applications.

**OVERVIEW**

These functions greatly simplify the tasks of the designer, allowing development efforts to be concentrated on the application, instead of on real-time coordination, task management problems, and complicated system generations. ZRTS provides a modular and flexible development tool that serves as a versatile base for Z8000 system designs. The Kernel requires only 4K bytes of either PROM or RAM memory, thus allowing configurations for a wide variety of target systems, while producing a memory-efficient, cost-effective end product.
FUNCTIONAL DESCRIPTION

The Concepts. ZRTS is both easy-to-learn and easy-to-use. Only a few simple concepts need to be understood before designing begins.

Tasks. Tasks are the components comprising a real-time application. Each task is an independent program that shares the processor with the other tasks in the system. Tasks provide a mechanism that allows a complicated application to be subdivided into several independent, understandable, and manageable units.

Semaphores. Semaphores provide a low overhead facility for allowing one task to signal another. Semaphores can be used for indicating the availability of a shared resource, timing pulses or event notification.

Exchanges and Messages. Exchanges and Messages provide the mechanism for one task to send data to another. A Message is a buffer of data, while an Exchange serves as a mailbox at which tasks can wait for Messages and to which Messages are sent and held.

The ZRTS Kernel. The Kernel is the basic building block of ZRTS and performs the management functions for tasks, semaphores, the real-time clock, memory and interrupts. The Kernel also provides for task-to-task communications via Exchanges and Messages. All requests for Kernel operations are made via system call instructions with parameters in registers, according to the standard Zilog calling conventions.

Task Management. One of the main activities of the Kernel is to arbitrate the competition that results when several tasks each want to use the processor. Each task has a unique task descriptor that is managed by the Kernel. The data contained in the descriptor include the task name, priority, state and other pertinent status information. ZRTS supports any number of tasks, limited only by the memory available to accommodate the task descriptors and stacks.

The Kernel maintains a queue of all active tasks on the system. Each task is scheduled for processor time based on its priority. The highest-priority task is ready to run gains control of the CPU; other tasks are queued. Tasks can be prioritized up to 32767 levels, with round-robin scheduling among tasks with the same priority.

Tasks can run either segmented or non-segmented code, in either normal or system mode. The numerous operations that may be performed on tasks are listed in Table 1.

<table>
<thead>
<tr>
<th>TABLE 1.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TASK MANAGEMENT</strong></td>
</tr>
<tr>
<td>T__Census</td>
</tr>
<tr>
<td>T__Create</td>
</tr>
<tr>
<td>T__Destroy</td>
</tr>
<tr>
<td>T__Lock</td>
</tr>
<tr>
<td>T__Reschedule</td>
</tr>
<tr>
<td>T__Resume</td>
</tr>
<tr>
<td>T__Suspend</td>
</tr>
<tr>
<td>T__Unlock</td>
</tr>
<tr>
<td>T__Wait</td>
</tr>
<tr>
<td><strong>SEMaphore MANAGEMENT</strong></td>
</tr>
<tr>
<td>Sem__Clear</td>
</tr>
<tr>
<td>Sem__Create</td>
</tr>
<tr>
<td>Sem__Destroy</td>
</tr>
<tr>
<td>Sem__Signal</td>
</tr>
<tr>
<td>Sem__Test</td>
</tr>
<tr>
<td>Sem__Wait</td>
</tr>
<tr>
<td><strong>CLOCK MANAGEMENT</strong></td>
</tr>
<tr>
<td>Clk__Delay__Absolute</td>
</tr>
<tr>
<td>Clk__Delay__Interval</td>
</tr>
<tr>
<td>Clk__Set</td>
</tr>
<tr>
<td>Clk__Time</td>
</tr>
<tr>
<td><strong>MEMORY MANAGEMENT</strong></td>
</tr>
<tr>
<td>Mem__Census</td>
</tr>
<tr>
<td>Alloc</td>
</tr>
<tr>
<td>Release</td>
</tr>
<tr>
<td><strong>INTER-TASK COMMUNICATION</strong></td>
</tr>
<tr>
<td>M__Acquire</td>
</tr>
<tr>
<td>M__Assign</td>
</tr>
<tr>
<td>M__Create</td>
</tr>
<tr>
<td>M__Destroy</td>
</tr>
<tr>
<td>M__Get__Descriptor</td>
</tr>
<tr>
<td>M__Read</td>
</tr>
<tr>
<td>M__Receive</td>
</tr>
<tr>
<td>M__Receive__Wait</td>
</tr>
<tr>
<td>M__Release</td>
</tr>
<tr>
<td>M__Reply</td>
</tr>
<tr>
<td>M__Send</td>
</tr>
<tr>
<td>M__Write</td>
</tr>
<tr>
<td>X__Create</td>
</tr>
<tr>
<td>X__Destroy</td>
</tr>
</tbody>
</table>
Semaphore Management. The Kernel provides semaphore management for synchronizing interacting tasks. A typical use of semaphores is to provide mutual exclusion of a shared resource. When a resource is to be used by only one task at a time, a semaphore with a counter of 1 controls the resource. Every task requiring the resource must first wait on the semaphore. Since the counter is 1, only one task will acquire the resource. The others will be queued on the semaphore and suspended until the semaphore is signaled that the resource is once again available. At that time, the first task on the semaphore queue will be made ready to run and can use the resource. After all tasks have acquired the resource and signaled the completion of their use, the semaphore returns to its original state with a counter of 1. Counters greater than one are useful when there are a number of similar resources, (i.e., three tape drives, four I/O buffers, etc.).

In ZRTS, a semaphore can count up to 32676 signals. The commands provided by the Kernel to manage semaphores are listed in Table 1.

Clock Management. ZRTS operates with a real-time clock that generates interrupts at a hardware-dependent rate. It is used for timed waits, timeouts, and round-robin scheduling. All times are given in number of ticks. The clock may be manipulated by the set of commands provided by the Kernel that are listed in Table 1.

Memory Management. Storage for ZRTS data structures is allocated either statically at system generation time, or dynamically at run time. Dynamic allocation occurs via a system call that specifies the attributes of the structure to be created and returns a name that can be used to refer to the structure. Memory is allocated in 256-byte increments, and can be released using a system call.

The storage allocator can also be called directly to obtain blocks of memory up to 64K bytes long, which can be used by the task for any purpose.

Interrupt Management. Interrupt-handling routines are provided for system calls, non-vectored interrupts and a hardware clock. The user must provide interrupt routines for whatever other vectored interrupts are included in the target system.

ZRTS can switch control to a task waiting for an external event within 500-microseconds after the occurrence of the event. This is based on the worst case with a 4MHz 28000. A more typical response time would be 250-microseconds. Quicker service of interrupts is possible through the use of user-written routines.

Inter-task Communication. The Kernel provides the capability for tasks to exchange information. This communication process occurs when one task sends a Message to an Exchange and another task receives the Message. A Message contains a length indicator, a buffer with a variable amount of data, and a code that identifies the Message type. The Exchange is a system data structure that consists of a queue for Messages sent but not yet received, a semaphore on which a task can wait for a Message, and an optional "pool" list from which Messages can be obtained quickly.

ZRTS provides several commands for inter-task communications. These are listed in Table 1.

ZRTS Configuration Language (ZCL). Since ZRTS's modular design leads to so many different configurations, a simple facility for generating the target operating system is a critical part of the ZRTS package. The ZRTS Configuration Language (ZCL) provides an easy-to-use means for generating the target system. Using ZCL, the designer can specify hardware information, software parameters, linkage information, and system data structures in high-level terms.
ZCL unburdens the user of the necessity to learn the details of the ZRTS internal structures. System data structures can be generated simply by specifying the appropriate parameters. The ZCL syntax is free-format with comments allowed to make the configuration commands more readable and maintainable.

ZCL input is comprised of a number of descriptive sections, each containing the details of the target operating system. The functions of these sections are described in Table 2. A sample system generation using ZCL is illustrated in Figure 1.

Development Environment. Application modules for ZRTS can be developed on any Zilog Z80 or Z8000-based development system and then down-loaded into a Zilog Development Module or a customized target system.

Subroutine libraries are provided for making ZRTS systems calls from programs written in PLZ/SYS, PLZ/ASM and C. Register usage in the system calls is compatible with the Zilog standard.

When using a Development Module, the Debugger can be used with the ZRTS modules for testing purposes. After the application is debugged, the system can be easily reconfigured for the final target hardware.

**ORDERING INFORMATION**

**Description**

ZRTS/8001 Zilog Real Time Software for the Z8001
ZRTS/8002 Zilog Real Time Software for the Z8002

**Prerequisites**

Zilog Development System
MCZ/1, PDS, ZDS Series or Z-LAB 8000 (Requires Software License)
Peripheral controller chip ties into 8- and 16-bit systems

Based on one-chip-microcomputer architecture, universal peripheral controller comes with either multiplexed or nonmultiplexed address and data lines, provides ROM-less and prototyping packages for product development

by John Banning and Pat Lin, Zilog Inc., Cupertino, Calif

The growing power of high-end microprocessors and the complexity of peripheral devices attached to them has given rise to a need for general-purpose distributed processors to handle increasingly complicated input/output activities. As such, these devices must themselves have respectable processing and I/O-manipulation abilities while being able to interact efficiently with high-end microprocessors. Ideally, they would also communicate with 8-bit midrange microprocessors and be low in cost.

Just such a processor has been based on the Z8 single-chip microcomputer. The Z-UPC universal peripheral controller combines the instruction and I/O capability of the Z8 with two versions of bus interfacing: the Z-UPC offers the Z-BUS interface found on the Z8000, and the Z-UPC/U provides a Z80-compatible interface. The Z-BUS interface allows flexible connection to larger microprocessor systems and control of distributed I/O peripheral functions by means of a multiplexed address and data bus. The Z80-bus interface provides easy interfacing with 8-bit microprocessors and others that employ nonmultiplexed address and data buses.

A logical organization

The UPC is partitioned into two functional blocks: the logic for interfacing with the host-processors, and the core microcomputer (Fig. 1). In the multiplexed (Z-BUS) version, communication between the host and the UPC takes place over the Z-BUS, which provides an 8-bit bidirectional address and data port (AD0-AD7) and a set of control lines (AS, DS, R/W, CS, WAIT). Also, under UPC program control, an optional daisy-chain interrupt structure—using request (INT), acknowledge (INTACK),

1. Microcomputer plus. The Z-UPC universal peripheral controller bases much of its architecture on the Z8 chip, to which it adds circuits at left for interfacing with a host processor. Shown is the Z-BUS-compatible version with multiplexed address and data lines.
2. File in. Of the UPC's 256-byte register file, 234 are general-purpose and can function as accumulators, buffers, pointers, or stack or index registers. The other 22 are specific pointers and registers, as well as status and control registers for the UPC's I/O facilities.

enable input (IE1), and enable output (IEO) lines—can be implemented. The microcomputer portion is based on the Z8 microcomputer architecture, whose central processing unit executes instructions averaging 2.2 microseconds each using a 4-megahertz clock source. The CPU's memory comprises 256 bytes of register-file random-access memory (which can be accessed directly by the host processor and the UPC), plus 2,048 bytes of read-only memory for program storage; other features include three I/O ports for device control, two timer/counters, and six vectored interrupts.

In addition to 234 general-purpose registers, the register file contains 19 control registers for configuring and controlling the Z-UPC's I/O facilities and three parallel I/O ports. The control registers both specify how the hardware is configured and should function and provide status information for it as well.

A multipurpose register file
All of the general-purpose registers can function as accumulators, data buffers, address pointers, and stack or index registers. All ports and control registers can be accessed by UPC instructions like any other register. Instructions can access the registers directly or indirectly with an 8-bit address field. However, a 4-bit addressing scheme, which makes use of a register pointer, can save memory and execution time. In this scheme the register file is divided into 16 register groups, each containing 16 contiguous locations. The register pointer determines which group is being accessed, and a 4-bit address field specifies the register within the group. This capability is especially useful to speed context switching.
Programs running on the UPC may communicate with those running on the master processor in a number of ways: under interrupt control (either by the UPC or master); by transfer of byte data through data, status, and command registers; and by transfer of blocks of data to and from the UPC's register file.

Three groups

The host CPU can directly access the 19 interface registers through the Z-BUS interface. As illustrated in Fig. 3, the registers are separated from the UPC's internal registers and divided into three groups:

- Those for interface status and control, which the master processor can access to control UPC-generated Z-BUS interrupts, to interrupt the UPC, and to control message transfer over the Z-BUS.
- Those for data, status, and control, which are mapped into 16 registers by the I/O register pointer, controlled by the UPC. That arrangement gives the master processor direct access to 16 registers and allows transfer of data, status information, or control commands between the UPC and master processor.
- Those for block access, used by the master to transfer blocks of data into or out of a buffer in the UPC's register file. The UPC has complete control over the placement and size of the buffer in its register file.

Each of the three types can be read or written by the master processor. Because the UPC software has complete control over how these register groups are mapped into the register space, the layout of data in the registers is independent of the host processor's software and protected from it.

The UPC and the master processor can operate completely independently of each other. The UPC can ignore the data transfer request from the master by setting a bit in its master-processor interrupt control register. Any attempt to transfer data from the master when this bit is set causes an error flag, which will cause an interrupt to the UPC (if interrupts are enabled).

I/O lines by the dozen

The UPC has 24 lines dedicated to input and output that are grouped into three 8-bit ports. Since the ports are mapped into the register file, I/O data can be directly manipulated by any instruction. Each port has a mode-control register, which allows the port functions to be changed during program execution; for example, each line of port 1 and port 2 can be individually configured as input or output under program control. Each port can have its output lines defined as push-pull or as open-drain drivers.

Port 3 is a multifunction port. It has four input and four output lines that can be used for I/O or control functions. The control functions available through this port include interlocked handshake lines for ports 1 and 2, interrupt request inputs, timer input and output, and Z-BUS interrupt control.

Timing and counting

To support timing and counting requirements of software routines, the UPC provides two 14-bit timer/counters, T0 and T1. Among the timer/counter functions that are easily implemented by the UPC are: interval delay timer, time-of-day clock, watchdog timer (as for refreshing dynamic memory), external event counting, variable pulse-train output, duration measurement for external
4. **Disk jockey.** The UPC makes a controller for a floppy-disk drive that actually stores the file system on chip. The host has only to specify the file name and the function to be performed. The 74LS299 shift register converts serial into parallel data, which enters port 1 on the UPC.

Events, and automatic delay after an external event.

Each timer/counter is divided into a 6-bit prescaler and an 8-bit counter and is driven by the internal UPC clock, divided by four. The internal clock for $T_1$ may be set up for gating or triggering by an external event, or it may be replaced by an external clock input. Each timer/counter may operate in either a single-pass or a continuous mode, so that after the last count either counting stops or the counter reloads and continues counting. The counter and prescaler registers may be altered individually while the timer/counter is running; software controls whether the new values are loaded immediately or when the end of count (EOC) is reached. The two timer/counters may be cascaded using the timer-input lines on port 3.

**Interrupting the controller**

To serve host or I/O-device requests quickly, the UPC provides six interrupts from eight different sources: three from ports, two from timer/counters, and three from the host-processor interface. All six interrupts may be individually or globally disabled. The interrupts are prioritized, with the interrupt-priority control register providing 48 different priority schemes for handling concurrent interrupts. What's more, the masking and prioritizing of the interrupts may be dynamically modified under program control.

The UPC's interrupts are vectored. When an interrupt occurs, the program counter and flags are pushed onto the stack, and control passes to one of six predetermined interrupt-handling routines. The routine is pointed to by an address that has been stored in the first 12 bytes of program memory. All of the interrupts are disabled after an interrupt is accepted. Interrupts can be nested by enabling them during the interrupt service routine; they are automatically enabled during the return from the routine.

The Z-UPC instruction set is compatible with the Z8 microcomputer instruction set (though the UPC's load-external-memory instruction is only available in the 64-pin RAM version of the Z8). This instruction set, comprising 129 instructions of 43 basic types and using six main addressing modes, speeds program execution and achieves byte efficiency. The types of data that it allows to be used include bits, binary-coded decimal digits, bytes, and 16-bit words.

**Z-BUS support**

The UPC can support the full Z-BUS interrupt structure, including daisy-chained priority resolution and vectored interrupt acknowledge. Using the interface control and status ports, the master processor has the full range of Z-BUS mechanisms for enabling or disabling Z-UPC interrupts, marking interrupts as being under service, clearing interrupts, setting interrupt vectors, and disabling interrupts from lower-priority devices.

A program running on the UPC can start the normal Z-BUS interrupt sequence (assuming interrupts are
enabled) by setting the interrupt-pending bit in its master-processor interrupt-control register. Once that is done, the UPC hardware automatically handles the Z-BUS interrupt protocol—including output of the interrupt vector, which is held in a separate control register.

The master can generate an interrupt to the UPC by setting the end-of-message flag in the master-processor interrupt-control port. In addition, UPC interrupts are generated from the master when an error condition, such as transferring a block of data that is too long, occurs.

**Block transfer within limits**

The UPC's block-access port is ideally matched to the block-I/O instructions of the Z8000 microprocessor. With a single block-I/O instruction the Z8000 can address the block-access port and transfer a string of bytes into or out of the UPC. Each access by the Z8000 to the block-access port causes a series of actions in the UPC involving its data-indirection register and limit-count control register. The data-indirection register points to the UPC register that is read or written when the master reads or writes the block-access port. After each such read or write, the data-indirection register is incremented and the limit-count register decremented. When the limit count reaches zero, any further transfers through the block-access port will abort and cause a length-error interrupt in the UPC.

**Ideas for application**

The intelligence and flexibility of the UPC make it suitable for a wide variety of applications and allow it to offload the host computer in several ways. The UPC is capable of doing intensive off-line calculations, for example, such as those for data encryption. Also, it can perform the various code conversions and data formatting that are usually required in processor-to-device and device-to-device communications. Furthermore, it can buffer the data and generate controls for I/O devices such as printers and keyboards.

Figure 4 illustrates the UPC's application as a disk controller. Here, a file system can actually be implemented in the controller itself. The host processor has only to specify the file name and the function to be performed on the file. Depending on the sector size, either the register file or the external RAM in the 64-pin RAM version can be used as for data buffering. The serial-to-parallel conversion is done by a 74LS299 shift register, and the data is transferred using handshake logic in and out of port 1. Also, cyclic-redundancy error checking can be done by the UPC.

A UPC software package is available for Zilog's PDS8000 development system that includes an assembler (PLZ/ASM), a linker, and an imager. PLZ/ASM is a free-format assembler that generates relocatable and absolute object-code modules. It makes provision for external symbolic references and global symbol definitions. Data declarations, control structures, and DO loops between them supply a structured approach to the task of assembly language programming.

A development board, similar to the one that is now available for the Z8, will also be available. It uses the 64-pin version of the UPC to prototype a UPC-based system. The code thus developed can later be transferred to the ROM in the mask-programmable 40-pin version of the UPC, or it can be made available in image form for downloading to the RAM version.

Two serial RS-232-C interfaces will allow the 11- by-14-inch board to be used alone with a cathode-ray-tube terminal or to be connected to one of Zilog's PDS or ZDS-1 series development systems. Cable connection to such a host system will permit the transfer of software from the host—where it is developed—to the board for testing. Included on the board is a 64-pin Z8, which serves as a program monitor for the UPC.

The board also contains 4-K bytes of 2716 erasable programmable ROM (for the monitor/debugger program) and 4-K bytes of 2114 static RAM. For the user who wishes to test a ROM-based version of his code, it also offers a socket for 4-K bytes of 2716 E-PROM that may be used in place of the RAM. The monitor/debugger software, comprising a terminal handler, a debugger, command interpreter, and an upload/download handler, provides the various commands necessary for control, I/O, and debugging.

A wrapped-wire area of 40 square inches accommodates additional customer interfaces or special application circuits. This arrangement allows for wide range of user applications.

**Aid in prototyping**

The Z-UPC Protopack—the ROM-less version of the standard Z-UPC, housed in a pin-compatible 40-pin package (Fig. 5) that carries a 24-pin socket to accommodate a 2716 E-PROM—is used for prototype development and preproduction of mask-programmed UPC-based applications. The 24-pin socket is equipped with 12 ROM address lines, 8 ROM data lines, and the necessary control lines for interfacing with the E-PROM for the first 2-K bytes of program memory.

Pin compatibility allows the user to design the printed-circuit board for a final 40-pin mask-programmed UPC and, at the same time, allows the use of UPC Protopack to build prototype and pilot-production units. When the final program is established, the user can then switch over to the 40-pin mask-programmed UPC for large-volume production. The Protopack is also useful in applications where masked ROM setup time and mask charges are prohibitive and program flexibility is desired.
Adapting Unix to a 16-bit microcomputer

Z-Lab software development system with text-processing utilities supports 16 users in C language, has 32-bit bus for future expansion

by Bruce Weiner and Douglas Swartz
Zilog Inc., Cupertino, Calif

In systems based on 16- and 32-bit microprocessors, software will account for the bulk of the development cost. As more and more logic is squeezed onto a single chip, the hardware development process is being simplified and its costs reduced. Simultaneously, however, more complex and, hence, more expensive software is going to be required.

Zilog’s recognition of this trend in computer technology (shown in Fig. 1) led to Zeus, the adaptation of the Unix operating system for the Z-Lab 8000 microcomputer [Electronics, Feb. 10, p. 33].

Both software and hardware played crucial roles in the creation of the Z-Lab development system. Components such as the Z8001 microprocessor, the memory_management unit (MMU), and the Z-bus backplane-interface (ZBI) bus structure were as critical to the potential of the system as was the software itself. Together, the Z-Lab and the Zeus operating system foster a software development environment that is a major step toward controlling the rapidly escalating software costs of microprocessor products.

Transporting a system

In selecting an operating system, there were two options: writing a new one from scratch or transporting an existing one to the Z8001. The decision was made to transport one—provided it was possible to find an existing operating system that could be adapted quickly and was well-suited to software development.

The search for such an ideal software environment ultimately led to the Unix operating system. This system was selected for four reasons: it was designed specifically for software development and text processing; it had already been transported successfully to 16- and 32-bit computers; it had a large existing software base with applications pertinent to a development environment; and it had a large user base.

Skyrocketing software. As products use more sophisticated microprocessors, there is an increase in the amount of engineering effort required to write software. As hardware costs drop, software costs are becoming the major product development expense.

Reprinted with permission of Electronics.
What is Unix anyway?

As a general-purpose, multiuser, interactive operating system, Unix offers facilities that are seldom found even on larger mainframes. Through its hierarchical file structure, any file can be traced back to a single root directory, thereby facilitating the management of mass memory. In addition, peripheral device handlers, files, and interprocess communications are all compatible with each other, simplifying program development. Since one major goal of Unix is to increase programmer productivity by providing a responsive working environment, Unix includes a vast library of utilities ranging from spelling correction routines to various compiler compilers and supports over a dozen languages, including the language C in which it is written.

However, the most important role of any operating system is still managing the mass-storage files in which all the programs and data reside. Unix imposes no particular structure upon the content of these ordinary files. Instead, it distinguishes between two kinds of special files even though they are treated identically by the programmer.

The first of these is a directory file that simply lists the names and vital statistics of other files. These other files may, in turn, be programs, data, or even other directories. This hierarchical structure results in an unusually well-organized system in which a file can be specified by its path name, which is a sequence of directory names separated by slashes that terminates with the desired file name. Thus, the same name can be used for files of similar function as long as they have different path names. For example, /Jones/Statistics and /Smith/Statistics both refer to a file named Statistics, but they are not the same file because they have different path names indicating they are listed in the unique directories Jones and Smith.

Just as directories are treated in the same manner as ordinary files by programmers, so are the second kind of special files—input/output calls. This distinction is the most unusual feature of Unix and one of its greatest advantages over other operating systems. These special files are read and written just like ordinary ones, except that the selected device is activated and the data is passed to it using whatever protocol is appropriate. Thus, programs can send data to, for example, a printer in exactly the same way they do to a disk file—except the name of the selected output unit is different.

Unix programs may communicate with each other in the same manner as I/O calls. The output of one program is directed to the input of another while each program thinks it is reading or writing a disk file. The communication link itself is called a pipe and can be created either by the program itself or interactively by the programmer. In this way, a group of related programs may pass data to each other in an extremely efficient manner.

Another feature of Unix is its ability to safeguard original programs. Before a program is executed, a fork, or replicate, operation copies the program, including the code, register values, open files, current directory and the like, into memory. The replicated process is executed, ensuring that the original is never lost or scrambled, in case execution does not take place properly.

Perhaps the most visible portion of Unix is the shell, or fundamental control program, which functions as the primary interface with the system user. As a command language, it offers the programmer a productive working environment. Multitasking permits programs to be started without loss of control of the console. Special command files may be set up so that any sequence of shell commands can be executed by a single user command. Commands can even be strung together at the console so that the results of one are fed directly to the input parameters of the next, in the same manner as pipes interfacing programs.

-R. Colin Johnson

When transported, the Unix operating system was enhanced in several ways so that the Z8001 implementation might run more reliably. For example, in the standard Unix operating system, nothing prevents two users from simultaneously modifying a file so that one user can accidentally invalidate the other's changes. The Zeus operating system qualifies the three standard Unix file-opening modes (read, write, and read and write) with three access-control modes specifying what other users can do with the file.

The Zeus access control modes are shared, read-only, and exclusive. The shared mode, the standard Unix control mode, allows other users access to any file they desire. In the read-only mode, other users may access the file only for read operations. In the exclusive mode, other users may not access the file at all; the first user opening the file has exclusive access to it until the file is closed. Any attempted access to a file that violates these parameters results in a failure of that open operation.

Other enhancements

A full-screen text editor, called the visual editor, has been implemented in Zeus for cathode-ray-tube terminals. Its data base contains terminal-control information that permits full-screen editing for almost any combination of CRT terminals. The terminal data base can be updated by the user when adding new terminals to the system.

The editor lets the user display text files one page at a time and rapidly move the cursor on that page, inserting or deleting characters, words, lines, or groups of lines with a minimum number of keystrokes. Several additional features are available, such as cut-and-paste and word-wrap facilities.

Rebuilding the system

Another enhancement is the Sysgen program, which automatically rebuilds the Zeus system, letting the user tailor it to specific requirements. The user can add disk and tape drives or other input/output devices using the Sysgen program as well.

Several other utility programs are supplied with Zeus. Learn, an interactive program, teaches new users how to fully exploit Zeus's facilities; Mail lets users send messages to each other in postal format; Calendar automatically reminds users of events scheduled during the day when they sign on and begin using their terminal; Spell is a spelling-error detection program that uses a 25,000 word dictionary; and Man prints selected portions of the Zeus reference manual on the user's terminal. Over
60 other utilities are furnished with the Z-Lab.

Almost the entire Unix operating system and its application programs are written in C, a system implementation language. The key to transporting Unix software is a C compiler that generates code for the target system, in this case the Z8001-based Z-Lab 8000. Although C carries a certain level of machine independence, this does not mean that the entire Unix operating system can be transported by merely recompiling it. Most application programs, however, can be transported in this manner.

Seventh edition

Specifically, the Zeus operating system is Zilog's enhanced version of the seventh edition of the Unix operating system, which was modified by Bell Laboratories to eliminate explicit machine dependencies and ease its transportation to other computers. Some implicit machine dependencies, however, must of necessity remain in the Unix kernel. For this reason, transportation to Z-Lab is greatly simplified by creating hardware very similar to those architectural features implicit in this kernel.

The two major machine dependencies in the Unix kernel are the size of integers and pointers and the memory management capability required by Unix software. Both the Unix kernel and C assume that integers and pointers are the same size and that integer arithmetic can thus be performed on pointers. Examining the evolution of the Unix system sheds light on how this machine dependency was handled.

C originally was developed to write Unix, and Unix originally was written for Digital Equipment Corp.'s PDP-11 family of 16-bit minicomputers. Further, the seventh edition of the Unix system was written specifically for PDP-11 systems with separate code and data address spaces. Thus, microcomputer hardware that provides facilities similar to those of a minicomputer such as DEC's PDP-11/70 should minimize the transportation effort.

On the basis of this background information, the design team decided to run the Z8001 microprocessor in the nonsegmented mode for user processes and for almost all of the kernel. In a nonsegmented mode, programs use 16-bit addresses and are limited to a single 64-k-byte segment. This means that both integers and pointers are considered 16-bit quantities and therefore integer arithmetic can be performed on them.

Because the Z8000 family can support separate code and data address spaces, user and system programs may have as much memory as a PDP-11/70—128-k bytes, of which 64-k bytes are code and 64-k bytes are data. Furthermore, the Z8001's 24-bit addressing scheme can handle a total system memory as large as 16 megabytes. Because the Z-Lab 8000 can handle up to 1.5 megabytes of memory the need for swapping programs in and out of main memory is reduced, thereby minimizing response time when a large number of users are on the system.

Much of the existing Unix software base takes advantage of the operating system's dynamic allocation of memory. This system characteristic has had a major impact on the hardware design of Z-Lab.

Memory management

Figure 2 shows how a C program's data is laid out in memory. This stack starts at the highest 16-bit data address and grows toward lower addresses. Global data that is statically allocated starts at address 0 and of course does not grow.

The Unix kernel provides system calls that allow a process to dynamically request more data memory. This dynamic data area starts just above the global data and fills in the unused addresses up to stack.

Memory space located between the stack and the dynamic data area is not necessarily allocated to one or the other. The hardware must therefore detect a memory reference in the constantly changing gap between the two memory areas and make sure they do not overlap. When an invalid access is detected, the kernel can either allocate more memory or terminate the process, as appropriate.

To protect the memory areas from invalid access, Zilog's Z8010 memory management unit was selected for the Z-Lab processor board. The MMU relocates addresses so that programs can be placed anywhere in physical memory and keeps the system from being corrupted if a user's program runs amok.

Nonsegmented solution

If Z-Lab were running in segmented mode, the two data areas would be placed in separate data segments, and the MMU could detect address violations as well as the need for more memory. In a nonsegmented mode, however, memory references to both bear the same segment number, so detecting a memory reference in the gap must be accomplished in another way in order to prevent the dynamic data area and the stack from overlapping.

Although the segmented-mode solution could not be used, it did provide the foundation for a nonsegmented
solution, in which the references to the two dynamic data areas are made through two different MMUs. In Fig. 3, a simplified block diagram of Z-Lab's memory management architecture shows that there are separate MMUs for the code, as well as for the stack and data address spaces. The MMU select logic determines which one should be activated and guarantees that only one will be active at any given time.

The operating system sets the break register, the key element in determining whether the stack or the data MMU will be activated, pointing to the highest address in the dynamic data area. On every data reference to memory, address bits 8 through 15 from the Z8001 are compared to the value in the break register. Data addresses greater than or equal to the break value activate the stack MMU; data addresses less than the break value activate the data MMU. The MMU selection occurs quickly enough for no wait states to be required, even with a 6-megahertz Z8001.

**Integrating hardware and software**

The memory management design discussed above handles Unix software and nonsegmented Z8000 programs. In addition, the memory management architecture of the Z-Lab processor board can be modified under program control to support segmented user and system programs. Future software releases can thus take full advantage of the 16 megabytes of address space provided by the segmented Z8001.

The Z-Lab development project was approached as an integrated product-design effort. A broad-based project team was selected to facilitate close cooperation among the hardware, software, and mechanical engineers, and the memory management architecture thus developed by the team solved problems that could not have been solved independently by any of the individual groups.

Likewise, the various goals of the Z-Lab system could be attained only with an integrated approach to the hardware, software, and mechanical engineering aspects of the project. Of these goals, the first was to design a Unix-based system with enough flexibility and file-system integrity that users could configure and maintain it themselves. A second goal was a performance level that could comfortably support up to 16 users. The final one involved packaging the system for the office environment.

To best achieve these goals, the project team sought a system design with minimum power consumption and noise levels. Thus, the Z-Lab offers high-performance minicomputer power in a quiet and easily portable package that consumes only 325 watts. It has no special power requirements and no cooling requirements, if ambient temperature stays below 40°C.

Z-Lab system hardware was also designed for expandability. Using a moderate-sized printed-circuit board (approximately 9 by 11 inches) kept the hardware configuration compact while allowing enough board area for future Z-Lab products. A highly reliable two-piece connector, although slightly more expensive than the conventional one-piece card-edge connector, improved connection reliability and permitted more connections per inch of pc-board edge.

**Bus with a future**

A semisynchronous bus, the ZBI, was chosen for its high level of system performance and input/output interface. All Z8000 peripheral circuits interface with the bus simply, needing buffering only to attain the TTL drive levels required on the backplane. Z80 peripherals can also be attached to the bus by generating the required
Architectural planning. The Z-Lab development system uses the proprietary ZBI 32-bit bus, an error-correcting memory controller that communicates with the main memory over a separate high-speed bus, and both Winchester disk and cartridge tape controllers.

Z80 timing with simple interface logic.

The ZBI is a true 32-bit bus with the address and the data multiplexed on the same lines (Fig. 4). The bandwidth of the bus (8 megabytes per second) is sufficient for future high-speed 32-bit processors and for peripheral controllers as well.

The Z-Lab error-correcting memory controller (ECC) supports 8-, 16-, and 32-bit data transfers, performing 32-bit error correction with the aid of seven extra syndrome random-access memories that hold the correction bits for every 32 data RAMS. The ECC communicates with its memory array cards over a very high-speed dedicated memory bus.

Maximizing memory capacity

All timing and refresh circuitry on the controller is centralized, maximizing memory capacity in the system. In addition to a maximum of 1.5 megabytes of ECC memory in the processor module enclosure, the Z-Lab unit has slots for the processor, cartridge tape controller, and Winchester disk controller cards as well.

Two of the Z-Lab's three peripheral controllers are intelligent, using Z80B 6-MHz microprocessors. This offers three distinct advantages.

First, device control chores are offloaded from the main processor. The operating system thus can communicate with the peripheral controllers using high-level commands that let the peripheral controllers work in parallel with the main processor. For example, Z-Lab can issue simultaneous reads or writes to more than one disk drive; the disk controller keeps track of head position, sector position, and data transfer.

Secondly, the intelligent peripheral controllers can perform self-diagnostics on power-up or on command, thus certifying to the host processor with a high degree of certainty that they are functioning correctly before processing begins.

Finally, product maintenance and upgrading is simplified by using firmware. As information is gathered on how the operating system interacts with the disk under different program mixes, the Winchester disk controller can be easily "tuned" for higher performance by altering the firmware.

Initial board set

The Z-Lab board set consists of:

- A processor board containing eight serial channels with programmable bit-rate, a parallel printer interface for either Centronics or Data Products-type printers, a memory management subsystem that supports either segmented or nonsegmented user processes, and read-only memory containing the bootstrap software and power-up diagnostics.
- An ECC memory controller that supports 32-bit error correction for up to 16 256-K-byte memory array cards. This board contains detection and reporting logic for uncorrectable errors and error-logging logic for correctable errors.
- One or more 256-K-byte memory array cards using high-speed 16-K dynamic RAMS.
- An intelligent cartridge tape controller that handles up to four tape drives for file archiving or for backup of the entire system.
- An intelligent Winchester disk controller that supports up to four 24-megabyte 8-in. Winchester disk drives.
- An optional serial I/O controller board that supports an additional eight serial lines and an additional printer port.

Several other subsystems will be offered with Z-Lab in the near future. An expansion chassis will increase the number of card slots in the unit from 10 to 20, the maximum number a ZBI bus can support, for constructing very large systems.

Another offering will be a compatible 40-megabyte Winchester drive (40- and 24-megabyte drives can be mixed on the system's Winchester controller). Zilog also will offer an intelligent serial controller that can perform direct-memory-access transfers to and from main memory, which will help improve system performance by reducing the amount of time that must be spent by the processor in servicing terminal interrupts.
Devotees of Unix, the operating system whose responsiveness has been compared to that of a well-tuned sports car, are adding to their number almost daily. This rapid expansion of the user base of Unix, developed at Bell Laboratories and licensed by Western Electric Co., has been spurred by the emergence of user-transparent versions made for computers ranging in size from the likes of IBM System 370 mainframes down to Z80-based 8-bit microcomputer systems.

Item: Texas Instruments Inc., Dallas, long known for its comprehensive software development system, is planning to implement Unix through a subcontract with a third-party software house.

Item: Lifeboat Associates, a leading 8-bit software publisher in New York, has just signed an exclusive marketing contract with Microsoft for end-user sales of its 16-bit Xenix-11 adaptation for PDP-11s.

Item: Intel Corp.'s Ada compiler for the iAPX 432 [Electronics, Feb. 24, p. 119] is written in Pascal on a VAX-11/780 under Unix. (When asked why Unix was used when the final compiler release will be under VMS, Nicole Allegre, Ada program manager for the Santa Clara, Calif., company, responds, "The programmers just really wanted to use it.")

Obeys orders. Those programmers at Intel are not alone. Their counterparts across the country have been taken by Unix's responsive software-development environment. Also, the language in which the original Unix is written, C, is one of the most respected of the structured languages extant [Electronics, May 8, 1980, p. 129].

Since Unix was developed on Digital Equipment Corp. machines, it has been widely used on PDP-11 minicomputers for some time. However, now that Western Electric allows systems with only a few users to pay a special per-user royalty fee, it has become economical for commercial software houses to configure Unix for even inexpensive systems. An increasing number of original-equipment manufacturers and commercial software houses should start offering Unix for various other computer systems.

Unix is in fact making a strong bid to become a standard among operating systems for the new wave of 16-bit microsystems, though it faces stiff competition from the entrenched operating system family from Digital Research, Pacific Grove, Calif. When that company's 16-bit implementation of its MP/M becomes available, it will include many of the facilities that make Unix so desirable—plus CP/NET, which allows both 16- and 8-bit microsystems to share expensive peripherals. OEMs can look forward to a rich selection of system-level software packages from which to choose. Even the 8-bit microsystems are acquiring Unix-like capabilities without having to sacrifice CP/M capability.

Drawbacks. Unix is not without its critics. They say that the system cannot be used easily by clerical personnel and cite difficult operations, like rebuilding the linked list that describes the hierarchical file structure after a system crash. Some say that Unix does not provide adequate file-protection systems to make it completely trustworthy in commercial uses.

Such criticism stems from Unix's initial target: cooperative multiprogrammer software projects in which most of the users were professional computer specialists. That is why many of the facilities provided by it are specifically aimed at efficient
program development. On the other hand, Unix is probably best known for its document-preparation and management functions, which are often used by nonprogrammers. And with the addition of a good screen-oriented editor, like Zilog's visual editor, Unix offers a wide avenue of capability for professionals and nonprogrammers alike.

**New version.** One of the latest Unix versions is the Zeus adaptation by Zilog Inc. Cupertino, Calif., for its Z-Lab software development system using the Z8000 ([Electronics, March 24, p. 120]). And to be released next month to selected OEMs is the Z8000 version called Xenix from Microsoft in Bellevue, Wash. ([Electronics, March 24, p. 34]). Among the first of the OEMs is Codata of Sunnyvale, which is working on a floppy- and hard-disk-based microsystem that makes use of a Multibus-compatible central processing unit. Later this year, the 8086 version of Xenix is to be delivered to Altos Computer Systems of Santa Clara for its single-board 8086-based microsystem.

After that, Microsoft plans to release a 68000 version (as does Whitesmiths Ltd. of New York in an original implementation), with an eye to the iAPX-432 and the 16000 in an attempt to establish Xenix as the standard version of Unix for 16-bit microsystems. Not only is Microsoft dedicated to marketing Unix, but it is also dedicated to using it: all product development programming in its Consumer Products division is done in C on a PDP-11/70 under Unix and then transported to the target microsystem.

The first computer to which the operating system was transferred from the one on which it was developed was the Interdata 8/32. The Wollongon Group of Palo Alto, Calif., now offers Unix for the 8/32, as well as for the rest of Perkin-Elmer's 32-bit minicomputers (Perkin-Elmer having bought Interdata).

**The same.** In the Wollongon offering, a supreme attempt has been made to make this implementation virtually identical to the original as it appears to the user, in the interest of program portability and of preserving a common command language across Unix systems.

Unix is also available from Amdahl Corp. for its IBM 370 look-alike, the 470 mainframe, and even for a computer that is specially optimized for the C language—the C/70—from BBN Computer Corp. ([Electronics, Nov. 6, 1980, p. 46]). These, like the others, are licensed by Western Electric.

However, before the licensing procedures were changed to accommodate small systems, several software developers began work on Unix look-alikes. These user-transparent, yet original, implementation projects are now coming to fruition.

One that has been around for more than a year is Whitesmiths' Idris ([Electronics, March 24, 1981, p. 125]). Some of the newer ones are aiming at the 8-bit market to maintain compatibility with current software bases. Two, for Z80-based microsystems using the S-100 bus, come from Morrow Designs of Richmond, Calif., and Cromemco Inc. of Mountain View, Calif., respectively.

**Subtasks.** Morrow Designs' version, called μUNIX, runs CP/M as one task within its multiuser environment, thereby maintaining compatibility with CP/M software while gaining the conveniences of a user-transparent Unix. The emphasis throughout has been on compatibility and portability; μUNIX is written entirely in Whitesmiths' C, which is not supplied with the package. Cromemco's version runs the CDOS operating system as a subtask and maintains compatibility with that already extensive software base, including its new C compiler.

There is even a version, from Technical System Consultants Inc., for Southwest Technical Products Corp.'s 6809-based 128-K-byte microsystem. Called Uniflex, it is written entirely in assembly language and includes most of Unix's features; it supports both floppies and a 20-megabyte hard disk. The West Lafayette, Ind., firm will add a 68000 version soon and is looking to Ada, Pascal, and C for future high-level language projects.

Reprinted with permission of Electronics
<table>
<thead>
<tr>
<th>Region</th>
<th>Office Name</th>
<th>Address</th>
<th>Phone(s)</th>
<th>TWX</th>
</tr>
</thead>
<tbody>
<tr>
<td>West</td>
<td>Sales &amp; Technical Center</td>
<td>Zilog, Incorporated</td>
<td>10340 Bubb Road, Cupertino, CA 95014</td>
<td>(408) 446-9846</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Phone: (408) 446-9846; TWX: 910-338-7621</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sales &amp; Technical Center</td>
<td>Zilog, Incorporated</td>
<td>(617) 523-4222; TWX: 710-332-1726</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18023 Sky Park Circle</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Irvine, CA 92714</td>
<td>Phone: (714) 549-2891</td>
<td>TWX: 910-595-2803</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15643 Sherman Way</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Van Nuys, CA 91406</td>
<td>Phone: (213) 989-7485</td>
<td>TWX: 910-495-1765</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sales &amp; Technical Center</td>
<td>Zilog, Incorporated</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2918 N. 67th Place #2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Scottsdale, AZ 85251</td>
<td>Phone (602) 990-1977</td>
<td></td>
</tr>
<tr>
<td>Midwest</td>
<td>Sales &amp; Technical Center</td>
<td>Zilog, Incorporated</td>
<td>890 East Higgins Road</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Schaumburg, IL 60195</td>
<td>Suite 147</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Phone: (312) 885-8080</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TWX: 910-291-1064</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sales &amp; Technical Center</td>
<td>Zilog, Incorporated</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>25349 Chagrin Blvd.</td>
<td>665-5571</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Woodmere, OH 44122</td>
<td>Phone: (216) 831-2957</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>FAX: 216-831-2957</td>
<td></td>
<td></td>
</tr>
<tr>
<td>South</td>
<td>Sales &amp; Technical Center</td>
<td>Zilog, Incorporated</td>
<td>2711 Valley View, Suite 103</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dallas, TX 75234</td>
<td>Phone: (214) 243-6550</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Zilog, Incorporated</td>
<td>207</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Austin, TX 78757</td>
<td>Phone: (512) 453-3216</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1442 U.S. Hwy 19 South</td>
<td>Technical Center</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Clearwater, FL 33516</td>
<td>Zilog, Incorporated</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>110 Gibraltar Road</td>
<td>8011</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Horsham, PA 19044</td>
<td>Munich, West Germany</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Phone: (215) 441-8282</td>
<td>Telephone: (628) 36131</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TWX: 510-665-7077</td>
<td>Telex: 529110 Zilog d.</td>
<td></td>
</tr>
<tr>
<td>United Kingdom</td>
<td>Sales &amp; Technical Center</td>
<td>Zilog, Incorporated</td>
<td>2711 Valley View, Suite 103</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dallas, TX 75234</td>
<td>Phone: (214) 243-6550</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Zilog, Incorporated</td>
<td>207</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Austin, TX 78757</td>
<td>Technical Center</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1442 U.S. Hwy 19 South</td>
<td>Zilog, Incorporated</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Clearwater, FL 33516</td>
<td>8011</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Horsham, PA 19044</td>
<td>Munich, West Germany</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Phone: (215) 441-8282</td>
<td>Telephone: (628) 36131</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>TWX: 510-665-7077</td>
<td>Telex: 529110 Zilog d.</td>
<td></td>
</tr>
<tr>
<td>Japan</td>
<td>Sales &amp; Technical Center</td>
<td>Zilog, Japan KK</td>
<td>TBS Kaikan Bldg.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>3-3 Akasaka S-Chome</td>
<td>Minato-Ku, Tokyo 107</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Phone: (628) 36131</td>
<td>J22846</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Telex: ESSO EAST</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Zilog, Inc. 10460 Bubb Road, Cupertino, California 95014

Telephone (408) 446-4666  TWX 910-338-7621

Printed in USA