Copyright 1983 by Zilog, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of Zilog.

The information contained herein is subject to change without notice. Zilog assumes no responsibility for the use of any circuitry other than circuitry embodied in a Zilog product. No other circuit patent licenses are implied.
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 and Zilog microprocessor products. It includes technical articles, application notes, concept papers, and benchmarks. This book is the second of an expected series of 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 applications lead to end product designs with more features, more simply implemented, and at a lower system cost. It is hoped that 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.

<table>
<thead>
<tr>
<th>Z8 FAMILY</th>
<th>8-Bit Single-Chip Microcomputer, 2K/4K Bytes ROM and 144 Bytes RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z8601/Z8603/Z86LD1 MCU</td>
<td>Microcomputer Unit</td>
</tr>
<tr>
<td>Z8611/2/3 MCU</td>
<td>Microcomputer Unit</td>
</tr>
<tr>
<td>Z8671 MCU</td>
<td>Microcomputer Unit with BASIC Debug</td>
</tr>
<tr>
<td>Z8681/2</td>
<td>ROMless</td>
</tr>
<tr>
<td>Z8090/4 &amp; Z8590/4 Z-UPC Controller</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Z80 FAMILY</th>
<th>8-Bit General-Purpose Microprocessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z8400 CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>Z8410 DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>Z8420 PIO</td>
<td>Parallel I/O Controller</td>
</tr>
<tr>
<td>Z8430 CTC</td>
<td>Counter/Timer Circuit</td>
</tr>
<tr>
<td>Z8440/1/2 SIO</td>
<td>Serial I/O Controller</td>
</tr>
<tr>
<td>Z8470 DART</td>
<td>Dual Asynchronous Receiver/Transmitter</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Z80L FAMILY</th>
<th>Low-Power 8-Bit General-Purpose Microprocessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z8300 CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>Z8320 PIO</td>
<td>Parallel Input/Output</td>
</tr>
<tr>
<td>Z8330 CTC</td>
<td>Counter/Timer Circuit</td>
</tr>
<tr>
<td>Z8340 SIO</td>
<td>Serial Input/Output</td>
</tr>
<tr>
<td>Z8000 FAMILY</td>
<td>16-Bit General-Purpose Microprocessor</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>Z8001/2 CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>Z8003/4 Z-VMPU</td>
<td>Virtual Memory Processing Unit</td>
</tr>
<tr>
<td>Z8010 Z-MMU</td>
<td>Memory Management Unit</td>
</tr>
<tr>
<td>Z8015 Z-PMMU</td>
<td>Paged Memory Management Unit</td>
</tr>
<tr>
<td>Z8016 Z-DTC</td>
<td>Direct Memory Access Transfer Controller</td>
</tr>
<tr>
<td>Z8030 Z-SCC</td>
<td>Serial Communications Controller</td>
</tr>
<tr>
<td>Z8031 Z-ASCC</td>
<td>Asynchronous Serial Communications Controller</td>
</tr>
<tr>
<td>Z8036 Z-CIO</td>
<td>Counter/Timer and Parallel I/O Unit</td>
</tr>
<tr>
<td>Z8038 Z-FIO</td>
<td>FIFO I/O Interface Unit</td>
</tr>
<tr>
<td>Z8060 Z-FIFO</td>
<td>Z-FIFO Buffer Unit and FIO Expander</td>
</tr>
<tr>
<td>Z8065 Z-BEP</td>
<td>Burst Error Processor</td>
</tr>
<tr>
<td>Z8068 Z-DCP</td>
<td>Data Ciphering Processor</td>
</tr>
<tr>
<td>Z8500 FAMILY</td>
<td>Universal Peripherals</td>
</tr>
<tr>
<td>Z8530 SCC</td>
<td>Serial Communications Controller</td>
</tr>
<tr>
<td>Z8531 SCC</td>
<td>Asynchronous Serial Communications Controller</td>
</tr>
</tbody>
</table>
# Table of Contents

## Z8 Family
- Z8 Subroutine Library .............................................. 1-3
- Z8 MCU Test Mode .................................................. 1-53
- Build a Z8-Based Control Computer with BASIC .................. 1-57
- Z8671 Seven-Chip Computer ........................................ 1-77
- A Single-Board Terminal Using the Z8590 Universal Peripheral Controller .................................................. 1-85

## Z80 Family
- Z80 CPU vs. 6502 CPU Benchmark Report .......................... 2-3
- Integrating 8-Bit DMA to 16-Bit System Tutorial ................ 2-23
- Interfacing Z80 CPUs to the Z8500 Peripheral Family ........... 2-29

## Z8000 Family
- Z80 Memory Expansion for the Z800 ............................... 3-3
- On-Chip Memory Management Comes to 8-Bit Microprocessors ... 3-15
- 8- and 16-Bit Processor Family Keeps Pace with Fast RAMs. .... 3-25

## Z80000 Family
- Cost-Effective Memory Selection for Z8000 CPUs ................ 4-3
- Benchmark Report: Z8000 vs. 68000 vs. 8086 ...................... 4-9
- Operating System Support - The Z8000 Way ....................... 4-21
- A Performance Comparison of Three Contemporary 16-Bit Microprocessors .................................................. 4-27
- 16-Bit Microprocessors Get a Boost from Demand-Paged MMU .......................... 4-39
- Segmentation Advances Microcomputer Memory Addressing ....... 4-45
- Initializing the Z8001 CPU for Segmented Operation with the Z8010 MMU .................................................. 4-53
- Nonsegmented Z8001 CPU Programming .............................. 4-59
- Calling Conventions for the Z8000 Microprocessor ................ 4-67
- Fast Block Moves with the Z8000 CPU .............................. 4-75
- Character String Translation: Z8000 vs. 68000 vs. 8086 ........ 4-79
- Z8002 CPU Small Single-Board Computer ........................... 4-79
- Interfacing the Z8500 Peripherals to 68000 ....................... 4-93
- Interfacing the Z-BUS Peripherals to the 8086/8088 ............... 4-105
- Z8016/Z8000 DTC DMA Transfer Controller ....................... 4-113
- Initializing the CIO .................................................. 4-139
- Using SCC with Z8000 in SDLC Protocol ............................ 4-153
- SCC in Binary Synchronous Communication ........................ 4-165
- Z8530/Z8030 SCC Initialization: A Worksheet and Example .... 4-175
- The Z-FIO in a Data Acquisition Application ...................... 4-183
This application note describes a preprogrammed Z8601 MCU that contains a bootstrap to external program memory and a collection of general-purpose subroutines. Routines in this application note can be implemented with a Z8 Protopack and a 2716 EPROM programmed with the bootstrap and subroutine library.

In a system, the user's software resides in external memory beginning at hexadecimal address 0800. This software can use any of the subroutines in the library wherever appropriate for a given application. This application example makes certain assumptions about the environment; the reader should exercise caution when copying these programs for other cases.

Following RESET, software within the subroutine library is executed to initialize the control registers (Table 1). The control register selections can be subsequently modified by the user's program (for example, to use only 12 bits of Ports 0 and 1 for addressing external memory). Following control register initialization, an EI

<table>
<thead>
<tr>
<th>Control Register</th>
<th>Address</th>
<th>Initial Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMR</td>
<td>F1H</td>
<td>00H</td>
<td>TO and T1 disabled</td>
</tr>
<tr>
<td>P2M</td>
<td>F6H</td>
<td>FFH</td>
<td>P20-P27 : inputs</td>
</tr>
<tr>
<td>P3M</td>
<td>F7H</td>
<td>10H</td>
<td>P2 pull-ups open drain; P30-P33 : inputs; P35-P37 : outputs; P34 : DM</td>
</tr>
<tr>
<td>P01M</td>
<td>F8H</td>
<td>D7H</td>
<td>P10-P17 : AD0-AD7; P00-P07 : A8-A15; normal memory timing; internal stack</td>
</tr>
<tr>
<td>IRQ</td>
<td>FAH</td>
<td>00H</td>
<td>no interrupt requests</td>
</tr>
<tr>
<td>IMR</td>
<td>FBH</td>
<td>00H</td>
<td>no interrupts enabled</td>
</tr>
<tr>
<td>RP</td>
<td>FDH</td>
<td>00H</td>
<td>working register file 00H-0FH</td>
</tr>
<tr>
<td>SPL</td>
<td>FFH</td>
<td>65H</td>
<td>1st byte of stack is register 64H</td>
</tr>
</tbody>
</table>
instruction is executed to enable interrupt processing, and a jump instruction is executed to transfer control to the user's program at location 0812H. The interrupt vectors for IRQ0 through IRQ5 are rerouted to locations 0800H through 080FH, respectively, in three-byte increments, allowing enough room for a jump instruction to the appropriate interrupt service routine. That is, IRQ0 is routed to location 0800H, IRQ1 to 0803H, IRQ2 to 0806H, IRQ3 to 0809H, IRQ4 to 080CH, and IRQ5 to 080FH. Figure 1 illustrates the allocation of Z8 memory as defined by this application note.

The subroutines available to the user are referenced by a jump table beginning at location 0018H. Entry to a subroutine is made via the jump table. The 32 subroutines provided in the library are grouped into six functional classifications. These classifications are described below, each with a brief overview of the functions provided by each category. Table 2 defines one set of entry addresses for each subroutine in the library.

- Binary Arithmetic: Multiplication and division of unsigned 8- and 16-bit quantities.
- BCD Arithmetic: Addition and subtraction of variable-precision floating-point BCD values.
- Conversion Algorithms: BCD to and from decimal ASCII, binary to and from decimal ASCII, binary to and from hex ASCII.
- Bit Manipulations: Packs selected bits into the low-order bits of a byte, and optionally uses the result as an index into a jump table.
- Serial I/O: Inputs bytes under vectored interrupt control, outputs bytes under polled interrupt control. Options provided include: odd or even parity, BREAK detection, echo input editing (backspace, delete) auto line feed
- Timer/Counter: Maintains a time-of-day clock with a variable number of ticks per second, generates an interrupt after a specified delay, generates variable width, variable frequency pulse output.

The listings in the "Canned Subroutine Library" provide a specification block prior to each subroutine, explain the subroutine's purpose, lists the input and output parameters, and gives pertinent notes concerning the subroutines. The following notes provide additional information on data formats and algorithms used by the subroutines.

Figure 1. "ROMless Z8" Subroutine Library Memory Usage Map
1. Although the user is free to modify the conditions selected in the Port 3 Mode register (P3M, F7H), P3M is a write-only register. This subroutine library maintains an image of P3M in its register P3M__save (7F H). If software outside of the subroutine package is to modify P3M, it should reference and modify P3M__save prior to modification of P3M. For example, to select P32/P35 for handshake, the following instruction sequence could be used:

   OR    P3M__save, #04H
   LD    P3M, P3M__save

2. For many of the subroutines in this library, the location of the operands (source/destination) is flexible between register memory, external memory (code/data), and the serial channel (if enabled). The description of each parameter in the specification blocks tells what the location options are.

   • The location designation "in reg/ext memory" implies that the subroutine allows the operand to exist in register or in external data memory. The address of such an operand is contained in the designated register pair. If the high byte of that pair is 0, the operand is in register memory at the address held in the low byte of the register pair. Otherwise, the operand is in external data memory (accessed via LDE).

   • The location designation "in reg/ext/ser memory" implies the same considerations as above with one enhancement: if both bytes of the register pair are 0, the operand exists in the serial channel. In this case, the register pair is not modified (updated). For example, rather than storing a destination ASCII string in memory, it might be desirable to output the string to the serial line.

3. The BCD format supported by the following arithmetic and conversion routines allows representation of signed variable-precision BCD numbers. A BCD number of 2n digits is represented in n+1 consecutive bytes, where the byte at the lowest memory address (byte 0) represents the sign and post-decimal digit count, and the bytes in the n higher memory locations (bytes 1 through n) represent the magnitude of the BCD number. The address of byte 0 and the value n are passed to the subroutines in specified working registers. Digits are packed two per byte with the most-significant digit in the high-order nibble of byte 1 and the least-significant digit in the low-order nibble of byte n. Byte 0 is organized as two fields:

   Bit 7 represents sign:
   1 = negative;
   0 = positive.

   Bits 0-6 represent post-decimal digit count.

   For example:

   byte 0 = 05H = positive, with five post-decimal digits
   = 80H = negative, with no post-decimal digits
   = 90H = negative, with 16 post-decimal digits

4. The format of the decimal ASCII character string expected as input to the conversion routines "dascbcd" and "dascwrd" is defined as:

   ( + 1 - ) ( <digit> ) [ [ ( <digit> ) ] ]

   in which
   ( ) Parentheses mean that the enclosed times or can be omitted.
   [ ] Brackets denote that the enclosed element is optional.

   Table 3 illustrates how various input strings are interpreted by the conversion routines.

5. The format of the decimal ASCII character string output from the conversion routine "bcddassc" operating on an input BCD string of 2n digits is

   1 sign of character ( + 1 - )
   2n-x pre-decimal digits
   1 decimal point if x does not equal 0
   x post-decimal digits

6. The format of the decimal ASCII character string output from the conversion routine "wrdassc" is

   1 sign character (determined by bit 15 of input word)
   6 pre-decimal digits
   no decimal point
   no post-decimal digits
Table 2. Subroutine Entry Points

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>001B</td>
<td>divide</td>
<td>16/8 unsigned binary division</td>
</tr>
<tr>
<td>001E</td>
<td>div_16</td>
<td>16/16 unsigned binary division</td>
</tr>
<tr>
<td>0021</td>
<td>multiply</td>
<td>8x8 unsigned binary multiplication</td>
</tr>
<tr>
<td>0024</td>
<td>mult_16</td>
<td>16x16 unsigned binary multiplication</td>
</tr>
</tbody>
</table>

**Binary Arithmetic Routines**

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0027</td>
<td>bcdadd</td>
<td>BCD addition</td>
</tr>
<tr>
<td>002A</td>
<td>bcdsub</td>
<td>BCD subtraction</td>
</tr>
</tbody>
</table>

**BCD Arithmetic Routines**

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>002D</td>
<td>bcddaasc</td>
<td>BCD to decimal ASCII</td>
</tr>
<tr>
<td>0030</td>
<td>dascbcd</td>
<td>Decimal ASCII to BCD</td>
</tr>
<tr>
<td>0033</td>
<td>bcddwrd</td>
<td>BCD to binary word</td>
</tr>
<tr>
<td>0036</td>
<td>wrdbcd</td>
<td>Binary word to BCD</td>
</tr>
<tr>
<td>0039</td>
<td>bythasc</td>
<td>Binary byte to hexadecimal ASCII</td>
</tr>
<tr>
<td>003C</td>
<td>wrdhasc</td>
<td>Binary word to hexadecimal ASCII</td>
</tr>
<tr>
<td>003F</td>
<td>hascwrd</td>
<td>Hexadecimal ASCII to binary word</td>
</tr>
<tr>
<td>0042</td>
<td>wrddasc</td>
<td>Binary word to decimal ASCII</td>
</tr>
<tr>
<td>0045</td>
<td>dascwrd</td>
<td>Decimal ASCII to binary word</td>
</tr>
</tbody>
</table>

**Conversion Routines**

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0048</td>
<td>clb</td>
<td>Collect bits in a byte</td>
</tr>
<tr>
<td>004B</td>
<td>tmj</td>
<td>Table jump under mask</td>
</tr>
</tbody>
</table>

**Bit Manipulation Routines**

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>004E</td>
<td>ser_init</td>
<td>Initialize serial I/O</td>
</tr>
<tr>
<td>0051</td>
<td>ser_input</td>
<td>IRQ3 (receive) service</td>
</tr>
<tr>
<td>0054</td>
<td>ser_rlin</td>
<td>Read line</td>
</tr>
<tr>
<td>0057</td>
<td>ser_rabs</td>
<td>Read absolute</td>
</tr>
<tr>
<td>005A</td>
<td>ser_break</td>
<td>Transmit BREAK</td>
</tr>
<tr>
<td>005D</td>
<td>ser_flush</td>
<td>Flush (clear) input buffer</td>
</tr>
<tr>
<td>0060</td>
<td>ser_wlin</td>
<td>Write line</td>
</tr>
<tr>
<td>0063</td>
<td>ser_wabs</td>
<td>Write absolute</td>
</tr>
<tr>
<td>0066</td>
<td>ser_wbyt</td>
<td>Write byte</td>
</tr>
<tr>
<td>0069</td>
<td>ser_disable</td>
<td>Disable serial I/O</td>
</tr>
</tbody>
</table>

**Serial Routines**

<table>
<thead>
<tr>
<th>Address</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>005B</td>
<td>tod_i</td>
<td>Initialize for time-of-day clock</td>
</tr>
<tr>
<td>006F</td>
<td>tod</td>
<td>Time-of-day IRQ service</td>
</tr>
<tr>
<td>0072</td>
<td>delay</td>
<td>Initialize for delay interval</td>
</tr>
<tr>
<td>0075</td>
<td>pulse_i</td>
<td>Initialize for pulse output</td>
</tr>
<tr>
<td>0078</td>
<td>pulse</td>
<td>Pulse IRQ service</td>
</tr>
</tbody>
</table>

**Timer/Counter Routines**
7. Procedure name: ser_input

The conclusion of the algorithm for BREAK detection requires the Serial Receive Shift register to be cleared of the character currently being collected (if any). This requires a software wait loop of a one-character duration. The following explains the algorithm used (code lines 464 through 472, Part II):

1 character time = \[
\frac{(128 \times PREO \times TO) \text{ sec}}{\text{sec}} \times \frac{10 \text{ bit}}{\text{bit}} \times \frac{1 \text{ char}}{\text{char}} = \frac{1280 \times PREO \times TO}{\text{XTAL}} \text{ sec} \times \frac{1 \text{ char}}{\text{char}}
\]

A software loop equal to one character time is needed:

1 character time = \[
2 \frac{\text{sec}}{\text{XTAL cycle}} \times n \frac{\text{cycle}}{\text{loop}} = \frac{2n \text{ sec}}{\text{XTAL loop}}
\]

Solve for \(n\):

\[
\frac{(1280 \times PREO \times TO)}{\text{XTAL}} = 2n
\]

\(n = 640 \times PREO \times TO\)

The register pair SERhtime, SER1time was initialized during ser Init to equal the product of the prescaler and the counter selected for the baud rate clock. That is,

\[\text{SERhtime, SER1time} = \text{PREO} \times \text{TO}\]

The instruction sequence

\[
\text{inlop: ld } r\text{SERtmp}1, \#53 \text{ (6 cycles)}
\]

\[
\text{lpl: djnz rSERtmp1, lpl (12/10 cycles taken/not taken)}
\]

executes in

\[
6 + (52 \times 12) + 10 \text{ cycles} = 640 \text{ cycles}
\]

8. BREAK detection on the serial input line requires that the receive interrupt service routine be entered within a half-a-bit time, since the routine reads the input line to detect a true (=1) or false (=0) stop bit. Since the interrupt request is generated halfway through reception of the stop bit, half-a-bit time remains in which to read the stop bit level. Interrupt priorities and interrupt nesting should be established appropriately to ensure this requirement.

\[
\frac{1/2 \text{ bit time}}{\text{sec}} = \frac{(128 \times \text{PREO} \times \text{TO})}{\text{XTAL} \times 2}
\]

### Table 3. Decimal ASCII Character String Interpretation

<table>
<thead>
<tr>
<th>Input String</th>
<th>Sign</th>
<th>Pre-Decimal Digits</th>
<th>Post-Decimal Digits</th>
<th>Terminator</th>
</tr>
</thead>
<tbody>
<tr>
<td>+1234.567,</td>
<td>+</td>
<td>1234</td>
<td>567</td>
<td>,</td>
</tr>
<tr>
<td>+----+.789+</td>
<td>-</td>
<td></td>
<td>789</td>
<td>+</td>
</tr>
<tr>
<td>1234..</td>
<td>+</td>
<td>1234</td>
<td></td>
<td>.</td>
</tr>
<tr>
<td>4976-</td>
<td>+</td>
<td></td>
<td>4976</td>
<td>-</td>
</tr>
</tbody>
</table>

NOTE: The terminator can be any ASCII character that is not a valid ASCII string character.
PART_I MODULE

!'ROMLESS Z8' SUBROUTINE LIBRARY PART I

Initialize:  a) Port 0 & Port 1 set up to address
             64K external memory;
             b) internal stack below allocated
                RAM for subroutines;
             c) normal memory timing;
             d) IMR, IRQ, TMR, RP cleared;
             e) Port 2 inputs open-drain pull-ups;
             f) Data Memory select enabled;
             g) EI executed to 'unfreeze' IRQ;
             h) Jump to %0812.

Note: The user is free to modify the initial
      conditions selected for a, b, and c above,
      via direct modification of the Port 0 & 1
      Mode register (P01M, %F8).

The user is free to modify the conditions
selected in the Port 3 Mode register (P3M, %F7).
However, please note that P3M is a write-only
register. This subroutine library maintains
an image of P3M in its register P3M save (%7F).
If software outside of the subroutine package
is to modify P3M, it should reference and modify
P3M save, prior to modification of P3M. For
example, to select P32/P35 for handshake, use
an instruction sequence such as:

           OR     P3M_save, %04
           LD     P3M, P3M_save

This is important if the serial and/or timer/
counter subroutines are to be used, since these
routines may modify P3M.

1
Access to GLOBAL subroutines in this library should be made via a CALL to the corresponding entry in the jump table which begins at address $000F. The jump table should be referenced rather than a CALL to the actual entry point of the subroutine to avoid future conflict in the event such entry points change in potential future revisions.

Each GLOBAL subroutine in this listing is headed by a comment block specifying its PURPOSE and calling sequence (INPUT and OUTPUT parameters). For many of the subroutines in this library, the location of the operands (sources/destinations) is quite flexible between register memory, external memory (code/data), and the serial channel (if enabled). The description of each parameter specifies what the location choices are:

- The location designation 'in reg/ext memory' implies that the subroutine allows that the operand exist in either register or external data memory. The address of such an operand is contained in the designated register pair. If the high byte of that pair is zero, the operand is in register memory at the address given by the low byte of the register pair. Otherwise, the operand is in external data memory (accessed via LDE).

- The location designation 'in reg/ext/ser memory' implies the same considerations as above with one enhancement: if both bytes of the reg. pair are zero, the operand exists in the serial channel. In this case, the register pair is not modified (updated). For example, rather than storing a destination ASCII string in memory, it might be desirable to output such to the serial line.

!
CONSTANT

!Register Usage!

RAM_START :=_%7F

P3M save := RAM_START

TEMP_3 := P3M_save-1

TEMP_2 := TEMP_3-1

TEMP_1 := TEMP_2-1

TEMP_4 := TEMP_1-1

!The following registers are modified/referenced by the Serial Routines ONLY. They are available as general registers to the user who does not intend to make use of the Serial Routines!

SER_char := TEMP_4-1

SER_tmp2 := SER_char-1

SER_tmp1 := SER_tmp2-1

SER_put := SER_tmp1-1

SER_len := SER_put-1

SER_buf := SER_len-2

SER_imr := SER_buf-1

SER_cfg :=

!Serial Configuration Data

bit 7 : =1 => odd parity on

bit 6 : =1 => even parity on

(bit 6,7 = 11 => undefined)

bit 5 : undefined

bit 4 : undefined

bit 3 : =1 => input editing on

bit 2 : =1 => auto line feed enabled

bit 1 : =1 => BREAK detection enabled

bit 0 : =1 => input echo on

117 !

op := $80

ep := $40

ie := $08

al := $04

be := $02

ec := $01

SER_get := SER_cfg-1

SER_flg := SER_get-1

!Serial Status Flags

bit 7 : =1 => serial I/O disabled

bit 6 : undefined

bit 5 : undefined

bit 4 : =1 => parity error

bit 3 : =1 => BREAK detected

bit 2 : =1 => input buffer overflow

bit 1 : =1 => input buffer not empty

bit 0 : =1 => input buffer full

sd := $80

pe := $10

bd := $08

bo := $04

bne := $02

bf := $01

RAM_TMR := RAM_START-_10

SER1time := SER_flg-1
The following registers are modified/referenced by the Timer/Counter Routines ONLY. They are available as general registers to the user who does not intend to make use of the Timer/Counter Routines!

\[
\begin{align*}
\text{TOD tic} & := \text{RAM TMR-2} \\
\text{TOD imr} & := \text{TOD tic-1} \\
\text{TOD hr} & := \text{TOD imr-1} \\
\text{TOD min} & := \text{TOD hr-1} \\
\text{TOD sec} & := \text{TOD min-1} \\
\text{TOD tt} & := \text{TOD sec-1} \\
\text{PLS-1} & := \text{TOD tt-1} \\
\text{PLS tmr} & := \text{PLS-1-1} \\
\text{PLS-2} & := \text{PLS tmr-1} \\
\text{RAM END} & := \text{PLS 2} \\
\text{STACK} & := \text{RAM END} \\
\end{align*}
\]

Equivalent working register equates for above register layout:

\[
\begin{align*}
\text{rP3Msave} & := \text{R15} \\
\text{rTEMP_3} & := \text{R14} \\
\text{rTEMP_2} & := \text{R13} \\
\text{rTEMP_1} & := \text{R12} \\
\text{rTEMP Th} & := \text{R12} \\
\text{rTEMP_Tl} & := \text{R13} \\
\text{rTEMP^4} & := \text{R11} \\
\text{rSERchar} & := \text{R10} \\
\text{rSERtmp2} & := \text{R9} \\
\text{rSERtmp1} & := \text{R8} \\
\text{rSERtmpl} & := \text{R8} \\
\text{rSERtmph} & := \text{R9} \\
\text{rSERput} & := \text{R7} \\
\text{rSERlen} & := \text{R6} \\
\text{rSERbuf} & := \text{RR4} \\
\text{rSERbufh} & := \text{R4} \\
\text{rSERbufl} & := \text{RR5} \\
\text{rSERimr} & := \text{R3} \\
\text{rSERoflg} & := \text{R2} \\
\text{rSERget} & := \text{R1} \\
\text{rSERflg} & := \text{R0} \\
\text{rUSER1en} & := \text{R6} \\
\text{rUSER2en} & := \text{RR4} \\
\text{rUSERbuf} & := \text{R4} \\
\text{rUSERbufh} & := \text{RR5} \\
\text{rUSERtmp} & := \text{R3} \\
\text{rUSERoflg} & := \text{R2} \\
\text{rUSERget} & := \text{R1} \\
\text{rUSERflg} & := \text{R0} \\
\text{rUSER1en} & := \text{R6} \\
\text{rUSER2en} & := \text{RR4} \\
\text{rUSERbuf} & := \text{R4} \\
\text{rUSERbufh} & := \text{RR5} \\
\text{rUSERtmp} & := \text{R3} \\
\text{rUSERoflg} & := \text{R2} \\
\text{rUSERget} & := \text{R1} \\
\text{rUSERflg} & := \text{R0} \\
\text{rUSER1en} & := \text{R6} \\
\text{rUSER2en} & := \text{RR4} \\
\text{rUSERbuf} & := \text{R4} \\
\text{rUSERbufh} & := \text{RR5} \\
\text{rUSERtmp} & := \text{R3} \\
\text{rUSERoflg} & := \text{R2} \\
\text{rUSERget} & := \text{R1} \\
\text{rUSERflg} & := \text{R0} \\
\end{align*}
\]

Equivalent working register equates for above register layout:

\[
\begin{align*}
\text{RAM TMRr} & := \text{R15} \\
\text{rTODtic} & := \text{R13} \\
\text{rTODimr} & := \text{R12} \\
\text{rTODhr} & := \text{R11} \\
\text{rTODmin} & := \text{R10} \\
\text{rTODsec} & := \text{R9} \\
\text{rTODtt} & := \text{R8} \\
\text{rPLS 1} & := \text{R7} \\
\text{rPLS 2} & := \text{R6} \\
\text{rPLS_1} & := \text{R5} \\
\end{align*}
\]
EXTERNAL
ser_init  PROCEDURE
ser_input  PROCEDURE
ser_rlin  PROCEDURE
ser_rabs  PROCEDURE
ser_break  PROCEDURE
ser_flush  PROCEDURE
ser_wlin  PROCEDURE
ser_wabs  PROCEDURE
ser_wbyt  PROCEDURE
ser_disable  PROCEDURE
ser_get  PROCEDURE
ser_output  PROCEDURE
tod_i  PROCEDURE
tod-  PROCEDURE
pulse_i  PROCEDURE
pulse-  PROCEDURE

$SECTION PROGRAM
GLOBAL

!Interrupt vectors!

P 0000 0800
P 0002 0803
P 0004 0806
P 0006 0809
P 0008 080C
P 000A 080F

IRQ_0_ARRAY [1 word] := [$0800]
IRQ_1_ARRAY [1 word] := [$0803]
IRQ_2_ARRAY [1 word] := [$0806]
IRQ_3_ARRAY [1 word] := [$0809]
IRQ_4_ARRAY [1 word] := [$080C]
IRQ_5_ARRAY [1 word] := [$080F]
GLOBAL

<table>
<thead>
<tr>
<th>Jump Table</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP INIT</td>
<td>END</td>
</tr>
</tbody>
</table>

Copyright ARRAY [† BIT]: '(C)1980 ZILOG'

Subroutine Entry Points:

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>JUMP</td>
<td>ENTRY</td>
</tr>
</tbody>
</table>

Binary Arithmetic Routines:

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Division</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP divide</td>
<td>16/8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Division</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP div_16</td>
<td>16/16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Multiplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP multiply</td>
<td>8x8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Multiplication</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP mult_16</td>
<td>16x16</td>
</tr>
</tbody>
</table>

BCD Arithmetic Routines:

<table>
<thead>
<tr>
<th>Procedure</th>
<th>BCD Addition</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP bcdadd</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Procedure</th>
<th>BCD Subtraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP bcdsub</td>
<td></td>
</tr>
</tbody>
</table>

Conversion Routines:

<table>
<thead>
<tr>
<th>Procedure</th>
<th>BCD to Decimal ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP bcddasc</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Decimal ASCII to BCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP dascbcd</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Procedure</th>
<th>BCD to Binary Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP bcdwrd</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Binary Word to BCD</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP wrdbcd</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Binary to Hex ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP bythasc</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Bin. Word to Hex ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP wrdhasc</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Hex ASCII to Bin Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP hascwrd</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Dec ASCII to Bin Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP dasswrd</td>
<td></td>
</tr>
</tbody>
</table>

Bit Manipulation Routines:

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Collect Bits in a Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP clb</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Table Jump Under Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP tjm</td>
<td></td>
</tr>
</tbody>
</table>

Serial Routines:

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Initialize Serial I/O</th>
</tr>
</thead>
<tbody>
<tr>
<td>JP ser_init</td>
<td></td>
</tr>
</tbody>
</table>
P 0051 8D 0000* 305 JP ser_input !IRQ3 (receive) service!
306
P 0054 8D 0000* 307 JP ser_rlin !read line!
308
P 0057 8D 0000* 309 JP ser_rabs !read absolute!
310
P 005A 8D 0000* 311 JP ser_break !transmit BREAK!
312
P 005D 8D 0000* 313 JP ser_flush !flush (clear)
314
data input buffer!
315
P 0060 8D 0000* 316 JP ser_wlin !write line!
317
P 0063 8D 0000* 318 JP ser_wabs !write absolute!
319
P 0066 8D 0000* 320 JP ser_wbyt !write byte!
321
P 0069 8D 0000* 322 JP ser_disable !disable serial I/O!
323

P 006C 8D 0000* 324 !Timer/Counter Routines!
325 JP tod_i !init for time of day!
326
P 006F 8D 0000* 327 JP tod !tod IRQ service!
328
P 0072 8D 0000* 329 JP delay !init for delay interval
330
P 0075 8D 0000* 331 JP pulse_i !init for pulse output!
332
P 0078 8D 0000* 333 JP pulse !pulse IRQ service!
334
P 007B 335 END JUMP

P 007B 338 !Initialization!
339 INIT PROCEDURE
340 ENTRY

P 007B E6 F8 D7 341 LD P01M,#(2)11010111
342 !internal stack;
343 P3M_save,#(2)00010000
344 !P3M is write-only,
345 normal memory
346 timing !
347 LD P3M_save,#(2)00010000
348 !keep a copy in
349 P3M is write-only,
350 RAM for later
351 so keep a copy in
352 reference !
353 LD P3M, P3M_save !set up Port 3
354 !I
355 LD SPL,#STACK !stack pointer!
356 CLR TMR !reset timers!
357 CLR P2M,#FF !fall inputs!
358 CLR IRQ !reset int. requests!
359 CLR IMR !disable interrupts !
360 CLR RP !register pointer!
361 LD SER_flg,#%80 !serial disabled!
362 EI !globally enable
363 interrupts !
364 END INIT
Binary Arithmetic Routines

397 CONSTANT
398 div LEN := R10
399 DIVISOR := R11
400 dividend HI := R12
401 dividend LO := R13
402 GLOBAL

P 0099
divide PROCEDURE

Purpose = To perform a 16-bit by 8-bit unsigned binary division.

Input =
R11 = 8-bit divisor
RR12 = 16-bit dividend

Output =
R13 = 8-bit quotient
R12 = 8-bit remainder
Carry flag = 1 if overflow
= 0 if no overflow
R11 unmodified

ENTRY

P 0099 A9 7C 418 ld TEMP_1, div LEN !save caller's R10!
P 009B AC 08 419 ld div LEN,#8 !LOOP COUNTER!
P 420
421 !CHECK IF RESULT WILL FIT IN 8 BITS!
P 009D A2 BC 422 cp DIVISOR, dividend HI
P 009F BB 02 423 jr UGT, LOOP TCARRY = 0 (FOR RLC)!
P 424 !overflow!
P 00A1 DF 425 SCF !CARRY = 1
P 00A2 AF 426 ret
P 427
P 00A3 10 ED 428 LOOP: RLC dividend LO !DIVIDEND * 2!
P 00A5 10 EC 429 RLC dividend HI
P 00A7 7B 04 430 jr c,subt
P 00A9 A2 BC 431 cp DIVISOR, dividend HI
P 00AB BB 03 432 jr UGT, next TCARRY = 0!
P 00AD 22 CB 433 subj: SUB dividend HI, DIVISOR
P 00AF DF 434 SCF !TO BE SHIFTED INTO RESULT!
P 00B0 AA F1 435 next: djnz div LEN, LOOP !no flags affected!
P 436
437 !ALL DONE!
P 00B2 10 ED 438 RLC dividend LO !CARRY = 0; no overflow!
P 00B4 A8 7C 439 ld div LEN, TEMP_1 !restore caller's R10!
P 00B6 AF 441 ret
P 00B7 442 END divide
CONSTANT
d16_LEN := R7
dvsr_hi := R8
dvsr_lo := R9
rem_hi := R10
rem_lo := R11
quot_hi := R12
quot_lo := R13

P 00B7

div 16 PROCEDURE
Purpose = To perform a 16-bit by 16-bit unsigned binary division.
Input = RR8 = 16-bit divisor
RR12 = 16-bit dividend
Output = RR12 = 16-bit quotient
RR10 = 16-bit remainder
RR8 unmodified

ENTRY
ld TEMP,1,d16_LEN!save caller's R10!
ld d16_LEN,#10!LOOP COUNTER!
clr rem_hi!carry = 0!
clr rem_lo
dlp_16: rlc quot_lo
rlc quot_hi
rlc rem_lo
rlc rem_hi
jr c,subt_16
cp dvsr_hi,rem_hi
cp dvsr_lo,Fem_lo
jr ugt,Skp_16_
jr ult,subt_16
subt_16: sub rem_lo,dvsr_lo
sub rem_hi,dvsr_hi

ENTRY
ld TEMP,1mul LEN!save caller's R10!
ld mul LEN,#8!INIT HIGH RESULT BYTE!
cmp PRODUCT_HI,MULTIPLIER!no flags affected!
lm PRODUCT_HI,mul LEN-;LOOPl
R12 = product
R11 unmodified

ENTRY
ld TEMP,1,mul LEN!save caller's R10!
ld mul LEN,#9!18 BITS!
clr PRODUCT_HI!INIT HIGH RESULT BYTE!
rcf PRODUCT_HI!CARRY = 0!

ENTRY
ld TEMP,1,loop1: RRC PRODUCT_HI
rrc PRODUCT_LO
jr NC,NEXT
add PRODUCT_HI,MULTIPLIER
loop1: djnz mul LEN,loop1!restore caller's R10!
ret

ENTRY
mul PROCEDURE
Purpose = To perform an 8-bit by 8-bit unsigned binary multiplication.
Input = R11 = multiplier
R13 = multiplicand
Output = RR12 = product
R11 unmodified

ENTRY
ld TEMP,1,mul LEN!save caller's R10!
ld mul LEN,#9!18 BITS!
clr PRODUCT_HI!INIT HIGH RESULT BYTE!
rcf PRODUCT_HI!CARRY = 0!

ENTRY
ld TEMP,1,loop1: RRC PRODUCT_HI
rrc PRODUCT_LO
jr NC,NEXT
add PRODUCT_HI,MULTIPLIER
loop1: djnz mul LEN,loop1!restore caller's R10!
ret

ENTRY
mul PROCEDURE
Purpose = To perform an 8-bit by 8-bit unsigned binary multiplication.
Input = R11 = multiplier
R13 = multiplicand
Output = RR12 = product
R11 unmodified

ENTRY
ld TEMP,1,mul LEN!save caller's R10!
ld mul LEN,#9!18 BITS!
clr PRODUCT_HI!INIT HIGH RESULT BYTE!
rcf PRODUCT_HI!CARRY = 0!

ENTRY
ld TEMP,1,loop1: RRC PRODUCT_HI
rrc PRODUCT_LO
jr NC,NEXT
add PRODUCT_HI,MULTIPLIER
loop1: djnz mul LEN,loop1!restore caller's R10!
ret

ENTRY
mul PROCEDURE
Purpose = To perform an 8-bit by 8-bit unsigned binary multiplication.
Input = R11 = multiplier
R13 = multiplicand
Output = RR12 = product
R11 unmodified

ENTRY
ld TEMP,1,mul LEN!save caller's R10!
ld mul LEN,#9!18 BITS!
clr PRODUCT_HI!INIT HIGH RESULT BYTE!
rcf PRODUCT_HI!CARRY = 0!

ENTRY
ld TEMP,1,loop1: RRC PRODUCT_HI
rrc PRODUCT_LO
jr NC,NEXT
add PRODUCT_HI,MULTIPLIER
loop1: djnz mul LEN,loop1!restore caller's R10!
ret

ENTRY
mul PROCEDURE
Purpose = To perform an 8-bit by 8-bit unsigned binary multiplication.
Input = R11 = multiplier
R13 = multiplicand
Output = RR12 = product
R11 unmodified

ENTRY
ld TEMP,1,mul LEN!save caller's R10!
ld mul LEN,#9!18 BITS!
clr PRODUCT_HI!INIT HIGH RESULT BYTE!
rcf PRODUCT_HI!CARRY = 0!

ENTRY
ld TEMP,1,loop1: RRC PRODUCT_HI
rrc PRODUCT_LO
jr NC,NEXT
add PRODUCT_HI,MULTIPLIER
loop1: djnz mul LEN,loop1!restore caller's R10!
ret

ENTRY
mul PROCEDURE
Purpose = To perform an 8-bit by 8-bit unsigned binary multiplication.
Input = R11 = multiplier
R13 = multiplicand
Output = RR12 = product
R11 unmodified

ENTRY
ld TEMP,1,mul LEN!save caller's R10!
ld mul LEN,#9!18 BITS!
clr PRODUCT_HI!INIT HIGH RESULT BYTE!
rcf PRODUCT_HI!CARRY = 0!

ENTRY
ld TEMP,1,loop1: RRC PRODUCT_HI
rrc PRODUCT_LO
jr NC,NEXT
add PRODUCT_HI,MULTIPLIER
loop1: djnz mul LEN,loop1!restore caller's R10!
ret

ENTRY
mul PROCEDURE
Purpose = To perform an 8-bit by 8-bit unsigned binary multiplication.
Input = R11 = multiplier
R13 = multiplicand
Output = RR12 = product
R11 unmodified

ENTRY
ld TEMP,1,mul LEN!save caller's R10!
ld mul LEN,#9!18 BITS!
clr PRODUCT_HI!INIT HIGH RESULT BYTE!
rcf PRODUCT_HI!CARRY = 0!

ENTRY
ld TEMP,1,loop1: RRC PRODUCT_HI
rrc PRODUCT_LO
jr NC,NEXT
add PRODUCT_HI,MULTIPLIER
loop1: djnz mul LEN,loop1!restore caller's R10!
ret

ENTRY
mul PROCEDURE
Purpose = To perform an 8-bit by 8-bit unsigned binary multiplication.
Input = R11 = multiplier
R13 = multiplicand
Output = RR12 = product
R11 unmodified

ENTRY
ld TEMP,1,mul LEN!save caller's R10!
ld mul LEN,#9!18 BITS!
clr PRODUCT_HI!INIT HIGH RESULT BYTE!
rcf PRODUCT_HI!CARRY = 0!

ENTRY
ld TEMP,1,loop1: RRC PRODUCT_HI
rrc PRODUCT_LO
jr NC,NEXT
add PRODUCT_HI,MULTIPLIER
loop1: djnz mul LEN,loop1!restore caller's R10!
ret

ENTRY
mul PROCEDURE
Purpose = To perform an 8-bit by 8-bit unsigned binary multiplication.
Input = R11 = multiplier
R13 = multiplicand
Output = RR12 = product
R11 unmodified

ENTRY
ld TEMP,1,mul LEN!save caller's R10!
ld mul LEN,#9!18 BITS!
clr PRODUCT_HI!INIT HIGH RESULT BYTE!
rcf PRODUCT_HI!CARRY = 0!

ENTRY
ld TEMP,1,loop1: RRC PRODUCT_HI
rrc PRODUCT_LO
jr NC,NEXT
add PRODUCT_HI,MULTIPLIER
loop1: djnz mul LEN,loop1!restore caller's R10!
ret

ENTRY
mul PROCEDURE
Purpose = To perform an 8-bit by 8-bit unsigned binary multiplication.
Input = R11 = multiplier
R13 = multiplicand
Output = RR12 = product
R11 unmodified

ENTRY
ld TEMP,1,mul LEN!save caller's R10!
ld mul LEN,#9!18 BITS!
clr PRODUCT_HI!INIT HIGH RESULT BYTE!
rcf PRODUCT_HI!CARRY = 0!

ENTRY
ld TEMP,1,loop1: RRC PRODUCT_HI
rrc PRODUCT_LO
jr NC,NEXT
add PRODUCT_HI,MULTIPLIER
loop1: djnz mul LEN,loop1!restore caller's R10!
ret

ENTRY
mul PROCEDURE
Purpose = To perform an 8-bit by 8-bit unsigned binary multiplication.
Input = R11 = multiplier
R13 = multiplicand
Output = RR12 = product
R11 unmodified

ENTRY
ld TEMP,1,mul LEN!save caller's R10!
ld mul LEN,#9!18 BITS!
clr PRODUCT_HI!INIT HIGH RESULT BYTE!
rcf PRODUCT_HI!CARRY = 0!

ENTRY
ld TEMP,1,loop1: RRC PRODUCT_HI
rrc PRODUCT_LO
jr NC,NEXT
add PRODUCT_HI,MULTIPLIER
loop1: djnz mul LEN,loop1!restore caller's R10!
GLOBAL:

mult 16

PROCEDURE

Purpose = To perform an 16-bit by 16-bit unsigned binary multiplication.

Input = RR8 = multiplier

RR12 = multiplicand

Output = RQ10 = product (R10, R11, R12, R13)

RR8 unmodified

Zero FLAG = 0 if result > 16 bits

= 1 if result fits in 16 (unsigned) bits (RR12 = result)

ENTRY

P 00F6 79 7C
P 00F8 7C 11
P 00FA B0 EA
P 00FC B0 EB
P 00FE CF
P 00FF C0 EA
P 0101 C0 EB
P 0103 C0 EC
P 0105 C0 ED
P 0107 FB 04
P 0109 02 B9
P 010B 12 A8
P 010D 7A F0
P 010F 78 7C
P 0111 A9 7C
P 0113 A4 EB 7C
P 0116 AF
P 0117

ENTRY

ld TEMP 1,m16 LEN !save caller's R7!
ld m16 LEN,#17 !16 BITS!
clr prod_hi
clr prod_lo !init product!
rcf !CARRY = 0!
loop16: rrc prod_hi
rrc prod_lo !bit 0 to carry!
rrc mult_hi !multiplicand / 2!
rrc mult_lo
jr nc,next16
add prod_lo,plier_lo
adc prod_hi,plier_hi
next16: djnz m16 LEN,loop16 !next bit!
ld m16 LEN,TEMP 1 !restore caller's R7!
ld TEMP 1,prod_hi !test product...
or TEMP 1,prod_lo !...bits 31 - 16!
ret
END mult_16
The BCD format supported by the following arithmetic and conversion routines allows representation of signed magnitude variable precision BCD numbers. A BCD number of 2n digits is represented in n+1 consecutive bytes where the byte at the lowest memory address ('byte 0') represents the sign and post-decimal digit count, and the bytes in the next n higher memory locations ('byte 1' through 'byte n') represent the magnitude of the BCD number. The address of 'byte 0' and the value n are passed to the subroutines in specified working registers. Digits are packed two per byte with the most significant digit in the high order nibble and the least significant digit in the low order nibble of 'byte n'. 'Byte 0' is organized as two fields:

- bit 7 represents sign:
  - = 1 => negative
  - = 0 => positive
- bit 6-0 represent post-decimal digit count

For example:
- 'byte 0' = %05 => positive, with 5 post-decimal digits
- = %80 => negative, with no post-decimal digits
- = %90 => negative, with 16 post-decimal digits

CONSTANT

bcd LEN := R12
bcd_SRC := R14
bcd_DST := R15
GLOBR

P 0117

procedure bcdsub

Purpose = To subtract two packed BCD strings of equal length.

Input = R15 = address of destination BCD string (in register memory).
R14 = address of source BCD string (in register memory).
R12 = BCD digit count / 2

Output = Destination BCD string contains the difference.
Source BCD string may be modified.
R12, R14, R15 unmodified if no error
R13 modified.
Carry FLAG = 1 if underflow or format error.

ENTRY
xor @bcd_SRC, $80  ; complement sign of subtrahend!
if fall into bcdadd!

END bcdsub
GLOBAL

PROCEDURE

Purpose = To add two packed BCD strings of equal length.

dst <-- dst + src

Input = R15 = address of destination BCD string (in register memory).
R14 = address of source BCD string (in register memory).

R12 = BCD digit count / 2

Output = Destination BCD string contains the sum.
Source BCD string may be modified.

R12, R14, R15 unmodified if no error
R13 modified.

Carry FLAG = 1 if overflow or format error.

ENTRY

delete all leading pre-decimal zeroes!

ENTRY
721 !swap source and destination operands!

P 0176 BB 06 717 7r ugt,ba_5 !SRC > DST!
P 0178 BB 23 718 7r ult,ba_4 !SRC < DST!
P 017A DA FF 719 djnz R13,ba_6 !loop!
P 017C BB 1F 720 7r ba_4 !DIST > or = SRC!

P 017D E8 EC 721 !exchange complete!
P 017F 08 EC 722 ba_5: 1d R13,bcd_LEN

P 0180 DE 723 inc R13 !include flag/size byte!
P 0181 02 ED 724 add bcd_SRC,R13
P 0182 02 FD 725 add bcd_DST,R13
P 0185 00 EE 726 ba_7: dec bcd_SRC
P 0187 00 8C 727 dec bcd_DST

P 0189 E5 7C 728 ld TEMP_1,bcd_SRC
P 018C E5 7B 729 ld TEMP_4,bcd_DST
P 018F F5 7B EE 730 ld @bcd_SRC,TEMP_4
P 0192 F5 7C EF 731 ld @bcd_DST,TEMP_1 !one byte swapped!
P 0195 DA EE 732 djnz R13,ba_7

P 0197 08 70 733 ld R13,TEMP_2
P 0198 08 EE 734 djnz R13,oa

P 019D 50 70 OE 735 push R13
P 019D 70 AO 736 pop R13 !restore!
P 019D 50 70 EF 737 R13 =< TEMP_2

P 019F 24 ED 7D 738 !R13 = DST post decimal digit count
P 01A0 C0 7D 739 TEMP_2 = SRC post decimal digit count
P 01A1 00 8C 740 R13 < TEMP_2 !done already!
P 01A2 A0 8C 741 sub TEMP_2,R13
P 01A4 FB 09 742 rrc TEMP_2 !alignment offset!
P 01A6 D8 EE 743 jr nc,ba_8 !digits word aligned!
P 01A8 01 EE 744 rotate out least significant SRC post decimal digit!
P 01AA B0 7C 745 ld R13,bcd_SRC !dec post dec digit #1!
P 01AC D6 0485' 746 call rdr !determine if addition or subtraction!
P 01AF E5 7C 747 xor @R13 !get starting addresses!
P 01B0 B5 EE 748 xor TEMP_4,@bcd_DST !sign of SRC!
P 01B2 24 7D 749 get starting addresses!
P 01B3 D8 EE 750 ba_8: 1d TEMP_4,bcd_SRC !sign of DST!
P 01B5 09 EE 751 xor TEMP_4,@bcd_DST !sign of DST!
P 01B6 00 8C 752 !done already!
P 01B8 BB 45 753 sub R13,TEMP_2
P 01BB 02 ED 754 add bcd_SRC,R13
P 01BC 02 FC 755 push R13 !restore!
P 01BD 00 8C 756 add bcd_DST,bcd_LEN

P 01C0 CF 757 !ready!!!
P 01C1 E5 7C 758 rcf !carry = 0!
P 01C4 7B 80 759 caller = 0!
P 01C6 BB 05 760 ba_11: 1d TEMP_1,@bcd_DST !add or sub?!
P 01C7 BB 05 761 tm TEMP_4,#180 !add!
P 01C9 35 EE 7C 762 jr z,ba_9 !add!
P 01C9 35 EE 7C 763 sbc TEMP_1,bcd_SRC
P 01CC BB 03 764 jr ba_10
P 01CE 15 EE 7C 765 ba_9: 1ad TEMP_1,bcd_SRC
P 01DF 7C 766 ba_10: 1da TEMP_1
P 01E0 4C 7C 767 1d @bcd_DST,TEMP_1
P 01E1 00 EE 768 dec bcd_DST
P 01E2 00 EE 769 dec bcd_SRC
P 01E3 DA E5 770 djnz R13,ba_11

P 01E4 D8 7D 771 !propagate carry thru TEMP_2 bytes of DST!
P 01E6 BB 06 772 1d R13,TEMP_2!
P 01E7 BB 23 773 inc R13 !may be zero!
P 01E8 DA 02 774 djnz R13,ba_12
P 01E9 BB 05 775 jr ba_13
P 01E9 17 EF 00 776 ba_12: 1ad @bcd_DST,#0
P 01E9 17 EF 777 da @bcd_DST
P 01E9 E0 00 778 dec bcd_DST
P 01E9 FA F7 779 djnz R13,ba_12
780  !carry propagate complete!
781  ba_13:  jr  nc, ba_14  !done!
782  !Rotate out least significant post decimal DST
783  digit to make room for carry at high end!

P 01EE E5  EF  7C  784  ld  TEMP_1, @bcd_DST
P 01F1 56  7C  7F  785  and  TEMP_1, #7F
P 01F4 6D  0203'  786  jp  z, ba_err  !no post dec digits!
P 01F7 E6  7C  10  787  ld  TEMP_1, #10
P 01FA D8  EF  788  ld  R13, bcd_DST
P 01FC D6  0485'  789  call  rdr
P 01FF 01  EF  790  dec  @bcd_DST  !dec digit cnt!
P 0201 CF  791  ba_14:  rcf
P 0202 AF  792  ret
P 0203 DF  793

P 0204 AF  794  ba_err:  scf
P 0205  795  ret
P 0206  796  END  bcdadd
Conversion Routines

GLOBAL

bcdasc PROCEDURE

Purpose = To convert a variable length BCD string to decimal ASCII.

Input = RR14 = address of destination ASCII string (in reg/ext/ser memory).
        R13 = address of source BCD string (in register memory).
        R12 = BCD digit count / 2

Output = ASCII string in designated destination buffer.

Carry FLAG = 1 if input format error or serial disabled,
   = 0 if no error.

R12, R13, R14, R15 modified.

Input BCD string unmodified.

ENTRY

section .data

ptr eq 0EF8h

section .text

bcdasc:

push bca_SRC

call put

string (in reg/ext/ser memory).

END- bcdasc

wrdhasc PROCEDURE

Purpose = To convert a binary word to Hex ASCII.

Input = RR12 = source binary word.
        RR14 = address of destination ASCII string (in reg/ext/ser memory).

Note = All other details same as for bythasc.

ENTRY

section .data

ptr eq 0EF8h

section .text

wrdhasc:

call bythasc

!convert R12!

section .data

ptr eq 0EF8h

section .text

bythasc:

call put

string in designated destination!

END- bythasc

1-22
CONSTANT bna_SRC := R12
GLOBAL

bythasc PROCEDURE

Purpose = To convert a binary byte to Hex ASCII.

Input = RR14 = address of destination ASCII string (in reg/ext/ser memory).
R12 = Source binary byte.

Output = ASCII string in designated destination buffer.
Carry = 1 if error (serial only).
R14, R15 modified.

ENTRY
clr MODE !flag => binary to ASCII!
ld TEMP_2,#2
SWAP bna_SRC
ld TEMP_1,bna_SRC
and TEMP_1,#%0F
isolate low nibble!
ADD TEMP_1,#%30
convert to ASCII!
ADD TEMP_1,#%3A
>9?!
ult,skip !no!
SCF !in case error!
MODE,#1
input is BCD?!
NZ,bca ex !yes, error.!
ADD TEMP_1,#%07
input hex. adjust!
call put_dest !put byte in dest!
c,bca ex !error!
DEC TEMP_2 !loop till done!
jr nz,bca_go1 !done!

END bythasc
P 0284 procedure

Purpose = To convert a variable length BCD string to a signed binary word. Only pre-decimal digits are converted.

Input = R14 = address of source BCD string (in register memory).
R15 = BCD digit count / 2

Output = RR12 = binary word
Carry FLAG = 1 if input format error or dest overflow,
= 0 if no error.
R14, R15 modified.

ENTRY

P 0284 B0 EC 956 clr R12 !init destination!
P 0286 B0 ED 957 clr R13
P 0288 E5 EE 7B 958 ld TEMP 4, @bcd adr !get sign/post length!
P 028B 56 7B 7F 959 and TEMP 4, $7F !isolate post length!
P 028E 02 FF 960 add bcd cnt, bcd cnt # bcd digits!
P 0290 24 7B EF 961 sub bcd_cnt, TEMP 4 # pre-dec digits!
P 0293 7B 37 962 jr ult, bcd w2 !format error!
P 0295 E5 EE 7B 963 ld TEMP 4, @bcd adr !remember sign!
P 0298 E6 7E 02 964 bcd_w3: ld TEMP 3, #2 !digits per byte!
P 029B EE 965 inc bcd adr !src address!
P 029C E5 EE 7D 966 ld TEMP 2, @bcd adr !get next src byte!
P 029F A6 EF 00 967 bcd_w1: cp bcd_cnt, #0 !digit count = 0?!
P 02A2 6B 12 968 jr z, bcd w4 !conversion complete!
P 02A4 FO 7D 969 swap TEMP 2 !next digit!
P 02A6 E4 7D 7C 970 ld TEMP 1, TEMP 2 !accumulate in binary!
P 02A9 D6 042C 971 call bcd Bin !accumulate in binary!
P 02AC 7B 1E 972 jr c, bcd w2 !overflow or format err!
P 02AE 00 EF 973 dec bcd cnt !update digit count!
P 02B0 00 7E 974 dec TEMP 3 !next byte?!
P 02B2 EB 975 jr nz, bcd w1 !no. same.!
P 02B4 8B E2 976 jr bcd w3 !next byte!
P 02B6 DF 977 bcd_w4: scf !in case!
P 02B7 76 EC 80 978 tm R12, %80 !result > 15 bits?!
P 02BA EB 10 979 jr nz, bcd w2 !overflow!
P 02BC 76 7B 80 980 bcd_w5: tm TEMP 4, %80 !source negative?!
P 02BF 6B 0A 981 jr z, bcd w6 !no. done.!
P 02C1 6C EC 982 com R12 !
P 02C3 60 ED 983 com R13 !
P 02C5 06 ED 01 984 add R13, #1 !
P 02C8 16 EC 00 985 adc R12, #0 !RR12 two's complement!
P 02CB CF 986 bcd_w6: scf !carry = 0!
P 02CC AF 987 bcd_w2: ret
P 02CD 988 END bcdwrd
GLOBAL
PROCEDURE

Purpose = To convert a signed binary word to a variable length BCD string.

Input = R14 = address of destination BCD string (in register memory)
        R12 = source binary word
        R15 = BCD digit count / 2

Output = BCD string in destination buffer
        Carry FLAG = 1 if dest overflow
                = 0 if no error.

R12, R13, R14, R15 modified.

ENTRY

P 02CD

P 02EF 7E 80 1008

P 02F2 EE 80 1010 or @bcd adr, #$80 1iset result negative!

P 02D9 60 EC 1011 com R13

P 02D0 6E ED 01 1013 add R13, #1

P 02DE 6D EC 00 1014 adc R12, #0 !RR12 two's complement!

P 02E3 10 EE 1016 rlc R12 !bit 15 not magnitude!

P 02E5 EE 1017 inc bcd adr !update dest pointer!

P 02E6 EE 7C 1018 ld TEMP_1, bcd adr !dest byte count!

P 02E8 EE 7D 1019 ld TEMP_2, bcd_cnt !dest byte count!

P 02EA 04 EE 1020 add TEMP_1, bcd_cnt

P 02ED 00 EE 1021 dec TEMP_1 !dest end addr!

P 02EF EE 80 1022 wrd_b1: clr @bcd adr !initialize dest!

P 02F1 EE 1023 inc bcd adr

P 02F2 EE FB 1024 djnz bcd_cnt, wrd_b1

P 02F4 EE 7E 0F 1025 ld TEMP_3, #15 !source bit count!

P 02F7 EE 7E 1026 wrd_b3: push TEMP_3

P 02F9 EE 1027 rlc R13

P 02FB EE 1028 rlc R12 !bit 15 to carry!

P 02FD EE 7C 1029 ld bcd adr, TEMP_1 !start at end!

P 02FF EE 7D 102A ld bcd_cnt, TEMP_2 !dest byte count!

P 0301 EE 7E 1031 li (dest bcd string) $ (dest bcd string #2) + carry!

P 0304 EE 7E 1032 wrd_b2: ld TEMP_3, bcd adr !* 2 + carry!

P 0307 EE 73 1034 da TEMP_3

P 0309 EE 7E EE 1035 ld @bcd adr, TEMP_3 !next two digits!

P 030C EE EE 1036 dec bcd adr !restore src bit cnt!

P 0310 EE 7E 1037 djnz bcd_cnt, wrd_b2 !loop for all digits!

P 0312 EE 7B 04 1039 jr c, wrd_ex !dest. overflow!

P 0314 EE 7E 1040 dec TEMP_3

P 0316 EE DF 1041 jr nz, wrd_b3 !next bit!

P 0319 EE 1042 wrd_ex: ret

END  wrd_bcd
Purpose = To convert a variable length Hex ASCII string to binary.

Input = RR14 = address of source ASCII string (in reg/ext/ser memory).

Output = RR12 = binary word (any overflow high order digits are truncated without error).

Carry FLAG = 1 if input error (SER flag indicates cause) = 0 if no error

R14, R15 modified

Note = The ASCII input string processing is terminated with the occurrence of a non-hex ASCII character.

GLOBAL

PROCEDURE

Purpose = To convert a variable length Hex ASCII string to binary.

Input = RR14 = address of source ASCII string (in reg/ext/ser memory).

Output = RR12 = binary word (any overflow high order digits are truncated without error).

Carry FLAG = 1 if input error (SER flag indicates cause) = 0 if no error

R14, R15 modified

Note = The ASCII input string processing is terminated with the occurrence of a non-hex ASCII character.

ENTRY clr TEMP_3
cline R12
has_c1: call get_src !get input!
has_ex1: ret
jrc

ENTRY clr R12
clr R13
has_c1: call get_src !get input!
has_ex1: ret
jrc

ENTRY clr R12
clr R13
has_c1: call get_src !get input!
has_ex1: ret
jrc

ENTRY clr R12
clr R13
has_c1: call get_src !get input!
has_ex1: ret
jrc
Procedure: dascwrd

Purpose = To convert a variable length decimal ASCII string to signed binary.

Input = RR14 = address of source ASCII string (in reg/extser memory).

Output = RR12 = binary word
R8,R9,R10,R11 holds the packed BCD version of the result.
Carry FLAG = 1 if input packed BCD error (SER_flg indicates cause)
= 0 if no error
R14, R15 modified

Note = The ASCII input string processing is terminated with the occurrence of a non-decimal ASCII character.
Decimal ASCII string may be no more than 6 digits in length, else Carry will be returned.
Post decimal digits are not included in the binary result.

Purpose:
16 digits!
Itemp addr =
R8 thru R11!
Iconvert to bcd1
error!
Iconvert to binary!
Purpose = To convert a variable length decimal ASCII string to BCD.

Input = R13 = address of destination BCD string (in register memory).
        RR14 = address of source ASCII string (in reg/ext/ser memory).
        R12 = BCD digit count / 2

Output = BCD string in designated destination buffer (any overflow high order digits are truncated without error).

Carry FLAG = 1 if input error (SER flag indicates cause) or overflow R14, R15 modified.

Note = The ASCII input string processing is terminated with the occurrence of a non-decimal ASCII character.

ENTRY
  push dab_LEN !save!
P 0365 70 EC
  push dab_DST
P 0367 B1 ED
  das_g1: clr @dab_DST !init. destination!
P 0369 DE
  inc dab_DST
P 036A CA FB
  djnz dab-LEN,das_g1
P 036C 50 EC
  pop dab-LEN
P 0370 70 ED
  ld TEMP 3,#1 !for ver_asc!
P 0375 B0 7B
  clr TEMP-4 !bit 0 => digit seen;
  bit 1 => dec pt seen;
  bit 7 => overflow!

P 0377 D6 03DA' das_g2: call get_src !get input byte!
P 037A 7B 41
  jr c,dab_ex1 !serial error!
P 037C 56 7C 7F
  and TEMP ?,#7F !7-bit ASCII!
P 037F 76 7B 03
  tm TEMP-4,#03 !check status!
P 0382 EB 0F
  jr nz,das_g5 !sign char not valid!
P 0384 A6 7C 2B
  cp TEMP 1,',' !positive?!
P 0387 6B EE
  jr z,das_g2 !yes. no affect!
P 0389 A6 7C 2D
  cp TEMP 1,,'-1 !negative?!
P 038C EB 07
  jr nz,das_g4 !not sign char!
P 038E B7 ED 80
  xor @dab_DST,#00 !complement sign!
P 0391 8B E4
  jr das_g2 !get next input!
P 0393 5B 0A
  das_g5: jr mi,Gas_g6 !dec pt has been seen!
P 0395 A6 7C 2E
  das_g7: cp TEMP 1,,'. !is char dec pt?!
P 0398 EB 05
  jr nz,Gas_g6 !nope!.
P 039A 46 7B 03
  or TEMP 4,#03 !dec pt and digit seen!
P 039D B8 D8
  jr das_g2 !get next input!
P 039F D6 040D' das_g6: call ver_asc !is bod digit?!
P 03A2 7B 16
  jr c,dab ex !end conversion!.
P 03A4 46 7B 01
  or TEMP 4,#01 !digit seen!
P 03A7 D6 0463' das_g2: call rd1 !new digit to dest!
P 03AA EB 09
  jr nz,das_g7 !overflow!
P 03AC 76 7B 02
  tm TEMP 4,#02 !post dec digit?!
P 03AF 6B C6
  jr z,das_g2 !no. get next input!

1-28
P 03B1 21 ED 1198  inc  @dab_DST !inc post dec cnt!
P 03B3 8B C2 1199  jr  das_g2 !get next input!
P 03B5 46 7B 80 1200 das_g7: or TEMP_4,#%80 !set overflow!
P 03B8 8B BD 1201  jr  das_g2 !get next input!
P 03BA E4 7B FC 1203 dab_ex: ld FLAGS,TEMP_4 !carry = 0 or 1!
P 03BD AF 1204 dab_ex1: ret
P 03BE 1205 END! dasbcd

GLOBAL

P 03BE 1208 wrddasc PROCEDURE
1209 !*****************************************************************************
1210 Purpose = To convert a signed binary word to
1211 decimal ASCII
1212
1213 Input = RR12 = source binary word.
1214 RR14 = address of dest (in reg/ext/ser
1215 memory).
1216
1217 Output = Decimal ASCII in dest buffer.
1218 R8,R9,R10,R11 holds the packed BCD
1219 version of the result.
1220 R12, R13, R14, R15 modified.
1221 *****************************************************************************
1222 ENTRY

P 03BE 70 EE 1223 push R14 !save dest addr!
P 03C0 70 EF 1224 push R15
P 03C2 EC 08 1225 ld R14,#8
P 03C4 04 FD EE 1226 add R14,RP !R8,9,10 & 11 temp!
P 03C7 FC 03 1227 ld R15,#3 !temp byte length!
P 03C9 D6 02CD' 1228 call wrdbcd !convert input word!
P 03CC 50 EF 1229 pop R15
P 03CE 50 EE 1230 pop R14 !restore dest addr!
P 03D0 CC 03 1231 ld R12,#3 !length of temp!
P 03D2 DC 08 1232 ld R13,#8
P 03D4 04 FD ED 1233 add R13,RP !addr of temp!
P 03D7 8D 0205' 1234 jp bcdasc !convert to ASCII!
P 03DA 1235 END wrddasc
GLOBAL !for PART II only!

**get src** PROCEDURE

**Purpose** = To get source byte from reg/ext/ser memory into TEMP_1.

**Output** = Carry FLAG = 1 if error (serial) 0 if all ok

**TEMP_1** = source byte.

**RR14** = updated.

ENTRY

rcf !set good return code!

inc R14 !test R14 = 0!

djnz R14, get_s1 !src in ext memory!

inc R15 !test R15 = 0!

djnz R15, get_s2 !src in reg memory!

ser get! src in ser memory!

push R11 !save user's!

ld TEMP_1, R11 !get byte!

pop R11 !restore user's!

Incw RR14 !update src ptr!

ret

ENTRY

rcf !set good return code!

inc R14 !test R14 = 0!

src in ext memory!

inc R15 !test R15 = 0!

ser get !src in ser memory!

get_s1: push R11 !save user's!

get_s2: ld TEMP_1, R15 !get byte!

ld TEMP_1, @RR14 !update src ptr!

ret

ENTRY

inc R14 !test R14 = 0!

src in ext memory!

inc R15 !test R15 = 0!

ser get !src in ser memory!

get_s1: push R11 !save user's!

get_s2: ld TEMP_1, @RR14 !update src ptr!

ret

ENTRY

inc R14 !test R14 = 0!

src in ext memory!

inc R15 !test R15 = 0!

ser get !src in ser memory!

get_s1: push R11 !save user's!

get_s2: ld TEMP_1, @RR14 !update src ptr!

ret

ENTRY

inc R14 !test R14 = 0!

src in ext memory!

inc R15 !test R15 = 0!

ser get !src in ser memory!

get_s1: push R11 !save user's!

get_s2: ld TEMP_1, @RR14 !update src ptr!

ret

ENTRY

inc R14 !test R14 = 0!

src in ext memory!

inc R15 !test R15 = 0!

ser get !src in ser memory!

get_s1: push R11 !save user's!

get_s2: ld TEMP_1, @RR14 !update src ptr!

ret

ENTRY

inc R14 !test R14 = 0!

src in ext memory!

inc R15 !test R15 = 0!

ser get !src in ser memory!

get_s1: push R11 !save user's!

get_s2: ld TEMP_1, @RR14 !update src ptr!

ret
1291 CONSTANT
1292 MODE := TEMP_3
1293 char := TEMP_1
1294 INTERNAL
1295 !****************************
1296 PROCEDURE
1297 Purpose = To verify input character as valid
1298 hex or decimal ASCII.
1299 Input = TEMP_1 = 8-bit input
1300 TEMP_3 = 0 => test for hex,
1301 1 => test for decimal
1302 Output = Carry FLAG = 0 if no error
1303 1 if error.
1304 ENTRY
1305 P 040D 56 7C 7F and char,$7F !7-bit ASCII!
1306 P 0410 A6 7C 30 cp char,$'0' !range start: '0'!
1307 P 0415 7B 16 jr ult,ver_err !no good!
1308 P 0410 A6 7C 3A cp char,$'9'+1 !dec range end: '9'!
1309 P 0412 BB 20 jr ult,ver ok !all's well!
1310 P 0415 76 7E 01 tm MODE,$1 !dec or hex?!
1311 P 0410 EB 0B jr nz,ver err !no good!
1312 P 041F 56 7C OF and char,$NOT('a'-'A') !insure upper case!
1313 P 0422 A6 7C 41 cp char,$'A' !check A-F range!
1314 P 0425 7B 04 jr ult,ver err !no good!
1315 P 0427 A6 7C 47 cp char,$'F'+1 !end hex range!
1316 P 042A EF 1320 ver err: ccf !complement carry
1317 P 042B AF 1321 ver-err: ret
1318 P 042C 1322
1319 P 044A 04 7C ED add R12,TEMP_1 !10x + d!
1320 P 0457 50 7C pop TEMP_1 !restore stack!
1321 P 0461 AF ret
1322 P 0463 1361 END bcd_bin
1323 P 042C 56 7C 0F and TEMP_1,$0F !isolate digit!
1324 P 042F A6 7C 09 cp TEMP_1,$9 !verify valid!
1325 P 0432 BB 2D jr ugt,bcd_b1 !error!
1326 P 0437 DD 37 add R13,R13- !error!
1327 P 0436 12 CC 38 adc R12,R12 !2x!
1328 P 0438 7B 27 jr c,bod_b1 !overflow!
1329 P 043A 70 EC push R12 -
1330 P 043C 70 0A push R13
1331 P 0438 02 DD 42 add R13,R13
1332 P 0440 12 CC 43 adc R12,R12 !4x!
1333 P 0442 7B 19 jr c,bod_b2 !overflow!
1334 P 0444 02 DD 45 add R13,R13
1335 P 0446 12 CC 46 adc R12,R12 !8x!
1336 P 0448 7B 13 jr c,bod_b2 !overflow!
1337 P 044A 04 7C ED 48 add R13,TEMP_1 !18x + d!
1338 P 0450 7B 0B jr c,bod_b2 !overflow!
1339 P 0452 50 7C pop TEMP T
1340 P 0454 04 7C ED 51 add R13,TEMP_1
1341 P 0457 50 7C pop TEMP_1
1342 P 0459 14 7C EC 53 add R12,TEMP_1 !10x + d!
1343 P 045C AF 55 ret
1344 P 045D 50 7C 57 bcd_b2: pop TEMP_1
1345 P 045F 50 7C 58 pop TEMP_1 !error!
1346 P 0461 5D 0D pop bcd_b1: sof
1347 P 0462 AF ret
1348 P 0463 1361 END bcd_bin
1363 CONSTANT
1364 s_len := R12
1365 s adr := R13
1366 INTERNAL
1367 rdl PROCEDURE
1368 |************************************************************************|
1369 Rotate Digit Left
1370 1371 Input = R12 = BCD string length
1372 R13 = BCD string address
1373 TEMP_1 bit 3-0 = new digit
1374
1375 Output = BCD string rotated left one digit;
1376 new digit inserted in units position.
1377 TEMP_1 bit 3-0 = digit rotated out
1378 of high order digit position
1379 bit 7-4 = 0
1380 Zero FLAG = 1 if TEMP_1 <> 0
1381 R12, R13 unmodified
1382
1383 ENTRY
1384 P 0463 70 EC 1385 push s_len
1386 P 0465 02 DC 1387 add s adr,s_len      !address of units place!
1388 P 0467 F1 ED 1389 rdl_01: swap @s adr
1390 P 0469 E5 ED 7D 1391 id TEMP_2,@s adr
1392 P 046C 57 ED FO 1393 and @s adr,UF      !isolate digit!
1394 P 046F 56 7C 0F 1395 and TEMP_1,%OF     !isolate new digit!
1396 P 0472 45 ED 7C 1397 or TEMP_1,@s adr
1398 P 0475 F5 7C ED 1399 ld @s adr,TEMP_1  !save new byte!
1400 P 0478 E4 7D 7C 1401 ld TEMP_1,TEMP_2
1402 P 047B 00 ED 1403 deco @s adr          !back-up pointer!
1404 P 047D CA E8 1405 djnz s_len,rdl_01  !loop till done!
1406 P 047F 56 7C 0F 1407 and TEMP_1,%OF     !old high order digit!
1408 P 0482 50 EC 1409 pop s_len           !restore R12!
1410 P 0484 AF 1411 ret
1412 P 0485 1413 END rdl

1400 INTERNAL
1401 rdr PROCEDURE
1402 |************************************************************************|
1403 Rotate Digit Right
1404 1405 Input = R12 = BCD string length
1406 R13 = BCD string address
1407 TEMP_1 bit 7-4 = new digit
1408
1409 Output = BCD string rotated right one digit;
1410 new digit inserted in high order
1411 position.
1412 R12 unmodified
1413 R13 modified
1414
1415 ENTRY
1416 P 0485 70 EC 1417 push s_len
1418 P 0487 DE 1419 rdl_01: inc s adr
1420 P 0488 F1 ED 1421 swap @s adr
1422 P 048A E5 ED 7E 1423 ld TEMP_3,@s adr
1424 P 048D 57 ED 0F 1425 and @s adr,UF      !isolate digit!
1426 P 0490 56 7C FO 1427 and TEMP_1,%OF     !isolate new digit!
1428 P 0493 45 ED 7C 1429 or TEMP_1,@s adr
1430 P 0496 F5 7C ED 1431 ld @s adr,TEMP_1  !save new byte!
1432 P 0499 E4 7E 7C 1433 id TEMP_1,TEMP_3
1434 P 049C CA E9 1435 djnz s_len,rdl_01  !loop till done!
1436 P 049E 50 EC 1437 pop s_len           !restore R12!
1438 P 04A0 AF 1439 ret
1440 P 04A1 1441 END rdr

1-32
Bit Manipulation Routines

1460 CONSTANT
1461 tjm_bits := R12
1462 tjm_mask := R13
1463 GLOBAL

P 04A1

1464 clb PROCEDURE
1465 

1466 Purpose = To collect selected bits in a byte into adjacent bits in the low order end of the byte. Upper bits in byte are set to zero.

1470 Input =

1472 R12 = input byte
1473 R13 = mask. Bit 1 -> corresponding input bit is selected.

1474 Output =

1475 R12 = collected bits

1476 Note = For example:

1478 Input : R12 = %21110110
1479 R13 = %21000010

1480 Output : R12 = %200000010

1481

1482 ENTRY

1483 E6 7C 08
1484 1484 ld TEMP_1,#8
1485 P 04A4 clr TEMP_2
1486 P 04AA clr TEMP_2
1487 P 04A8 next1: rl tjm_bits
1488 P 04A8 rl tjm_bits
1489 P 04AA rr tjm_bits
1490 P 04AE rl tjm_bits
1491 P 04BB rlc TEMP_2
1492 P 04B2 no_select:
1493 dec TEMP_1
1494 P 04B4 jr nz,next1
1495 P 04B6 ld R12,TEMP_2
1496 P 04B8 ret
1497 P 04B9 clb

END clb
P 04B9

1499 CONSTANT
1500 tjm_tabh  :=  R14
1501 tjm_tabl  :=  R15
1502 tjm_tab  :=  RR14
1503 GLOBAL
1504 tjm PROCEDURE
1505 !********************************************************************************
1506 Purpose = To take a jump to a routine address
1507 determined by the state of selected
1508 bits in a source byte. A bit
1509 is 'selected' by a one in the
1510 corresponding position of a mask.
1511 The 'selected' bits are packed into
1512 adjacent bits in the low order end of
1513 the byte. This value is then doubled,
1514 and used as an index into the jump
1515 table.
1516
1517 Input =
1518    RR14 = address of jump table in
1519 program memory.
1520    R12 = input data
1521    R13 = mask
1522 ********************************************************************************
1523 ENTRY
1524 call clb !collect selected bits!
1525 add tjm_bits,tjm_bits !collected bits * 2!
1526 adc tjm_tabh,$0= > in case carry!
1527 adc tjm_tabl,tjm_bits !tjm_tab points to...
1528 lda tjm_mask,@tjm_tab !...table entry!
1529 ines tjm_tab
1530 ld@ tjm_tabl,tjm_mask !get table entry...
1531 ldt@ tjm_tabh,tjm_mask !...into tjm_tab!
1532
1533  &tjm_tab !bye!
1534
1535 END tjm
1536 END PART_I

0 errors
Assembly complete
ROMLESS Z8 SUBROUTINE LIBRARY PART II

1-35
IThe following registers are modified/referenced
by the Timer/Counter Routines ONLY. They are
available as general registers to the user
who does not intend to make use of the
Timer/Counter Routines!

The equivalent working register equates
for above register layout!

!register file %70 - %7F!

!register file %60 - %6F!

EXTERNAL

get src PROCEDURE
put dest PROCEDURE
multiply PROCEDURE

$SECTION PROGRAM
Serial Routines

164 CONSTANT
165 si_PTR := RR14
166 si_TMP1 := R11
167 si_TMP2 := R13
168 GLOBAL
169 ser_init PROCEDURE
170 !**************************************************************************
171 serial init
172
173 Purpose = To initialize the serial channel and
174 RAM flags for serial I/O. Serial
175 input occurs under interrupt control.
176 Serial output occurs in a polled mode.
177
178 Input = RR14 = address of parameter list in
179 program memory (if R14 = 0,
180 use defaults):
181 1 byte = Serial Configuration Data
182 (see definition of SER_cfg)
183 1 byte = IMR mask for nestable
184 interrupts
185 1 word = address of circular input
186 buffer (in reg/ext memory)
187 1 byte = Length of input buffer
188 1 byte = Baud rate counter value
189 1 byte = Baud rate prescaler value
190 (unshifted)
191
192 Output = Serial I/O operations initialized.
193 R11, R12, R13, R14, R15 modified.
194
195 Note = Defaults:
196 Input echo on
197 Input editing on
198 BREAK detection enabled
199 No parity
200 Auto line feed on
201 Input Buffer Address = SER_char
202 Input buffer length = 1 byte
203 Baud Rate = 9600 (assuming
204 XTAL = 7.3728 MHz)
205
206 The instruction at %0809 must result
207 in a jump to the jump table entry for
208 ser_input.
209
210 If BREAK detection is disabled, and a
211 BREAK occurs, it will be received as a
212 continuous string of null characters.
213
214 The parameter list is not referenced
215 following initialization.
216
217 ENTRY
218 inc R14 !use defaults?!
219 djnz R14,si 1 !no. given by caller.!
220 ld R14,#HT ser_def !address of default...
221 ld R15,#LO ser-def !... parameter list.!
222 si_1: ld si_TMP1,#SER_cfg
223 si_2: ld si_TMP2,#5
224 si_3: ldci si_TMP1,#si_PTR !get initialization...
225 si_4: djnz si_TMP2,si 2 !... parameters!
226 and SER_imr,#F7 !insure no self-nesting!

1-37
initialize Port 3 Mode Register for serial I/O!

P 0012 56 F1 FC 229 AND TMR, #%FC !disable TO!
P 0015 B8 72 230 ld si_TMP1, SER cfg !configuration data!
P 0017 56 EB 80 231 AND si_TMP1, #%80 !odd parity select!
P 001A 46 EB 40 232 OR si_TMP1, #%40 !P30/7 = Sin/Sout!
P 001D 56 7F 3F 233 AND P3M_save, #%3F !mask off old settings!
P 0020 44 EB 7F 234 OR P3M_save, si_TMP1 !new selection!
P 0023 E4 7F F7 235 LD P3M, P3M_save !to write-only register!

initialize TO!
P 0026 BC F4 238 ld si_TMP1, #TO
P 0028 C2 DE 239 ldc si_TMP2, @si_PTR !save counter!
P 002A C3 BE 240 ldi @si_TMP1, @si_PTR !init counter!
P 002C C2 BE 241 ldc si_TMP1, @si_PTR !get prescaler!
P 002E D6 0000* 242 call multiply !TO x PRE0!
P 0031 C9 6E 243 ld SER_time, R12 !save for BREAK...
P 0033 D9 6F 244 rd SER_time, R13 !...detection !
P 0035 90 EB 245 ri si_TMP1 !SHL 1!
P 0037 DF 246 sof !continuous mode!
P 0038 10 EB 247 rlc si_TMP1 !SHL 2!
P 003A B9 F5 248 ld PRE0, si_TMP1

initialize RAM flags and pointers!
P 003C 8F 250 DI !disable interrupts!
P 003D B0 71 251 clr SER_get !input buffer...!
P 003F B0 77 252 clr SER_put !...empty!
P 0041 B0 70 253 clr SER_flag !no errors!

initialize interrupts!
P 0043 56 FA E7 256 AND IRQ, #%E7 !clear IRQ3 & 4!
P 0046 56 FB EF 257 and IMR, #%EF !disable IRQ4 (xmt)!
P 0049 46 FB 08 258 or IMR, #%08 !enable IRQ3 (rcv)!
P 004C 9F 259 EI

!go!
P 004D 46 F1 03 260 or TMR, #%03 !load/enable TO!
P 0050 AF 262 ret
P 0051 263 END ser_init

Defaults for serial initialization!
P 0051 0F 00 266 ser_def RECORD [cfg _, _imr_ BYTE
P 0053 007A 01 269 buf _
P 0056 02 03 270 len_, ctrl_, pre_ WORD

[ec+al+ie+be, %00, SER_char, 1, %02, %03]
CONSTANT
rli_len := R13

GLOBAL

PROEDURE read line

Purpose = To return input from serial channel
up to 'carriage return' character or
maximum length requested or BREAK.

Input = RR14 = address of destination buffer
(in reg/ext memory)
R13 = maximum length

Output = Input characters is destination buffer.
RR14 = unmodified
R13 = length returned

Note = 1. Return will be made to the calling
program only after the requisite
characters have been received from
the serial line.

2. If input editing is enabled, a
'backspace' character will cause
the previous character (if any) in the
the destination buffer to be deleted;
a 'delete' character will cause all
previous characters (if any) in the
destination buffer to be deleted.

3. If parity (odd or even) is enabled,
the parity error flag (R14) will be set
if any character returned had a parity
error. (Bit 7 of each character may
then be examined if it is desirable to
know which character(s) had the error).

4. The status flags 'BREAK detected',
'parity error', and 'input buffer
overflow' will be returned
as part of R12, but will be cleared in
SER_stat.

5. The status flags: 'input buffer full'
and 'input buffer not empty' will be
updated in SER_stat.

ENTRY

clr TEMP_3
flag => read line!

P 005A 70 EE
push R14
!save original...

P 005C 70 EF
push R15
!...dest. pointer!

P 005E 70 ED
push rli_len
!...and length!

P 0060 D6 0170'
call ser_get
!get input character!

P 0063 7B 48
jr c.rli_3
!error!

P 0065 76 72 CO
tm SER_cTg,#op LOR ep !parity enabled?!

P 0066 6B 08
jr z,rli_1
!no!

P 0067 7C 80
tm TEMP_1,#%80
!parity error?!

P 006D 6B 03
jr z,rli_1
!no!

1-39
P 006F 46 70 10 339 or SER_flg,#pe !yes. set error flag!
P 0072 D6 0000* 340 rli_1: call put_dest !store in buffer!
P 0075 A6 7E 00 341 cp TEMP_3,#0 !read line?!
P 0078 EB 31 342 jr nz,rT_2 !no!
P 007A 56 7C 7F 343 and TEMP_1,#7F !ignore parity bit!
P 007D 76 72 08 344 tm SER_cfg,#ie input editing on?!
P 0080 6B 21 345 jr z,rT_9 !no.!!

P 0082 A6 7C 7F 346 !input editing!
P 0085 6B 3E 348 jr z,rT_6 !yes!
P 0087 A6 7C 08 349 cp TEMP_7,#08 !char = delete?!
P 008A EB 17 350 jr nz,rT_9 !no. continue!
P 008C 50 7C 351 pop TEMP_1 !get original length!
P 008E 70 7C 352 push TEMP_1
P 0090 A4 ED 7C 353 cp TEMP_1,rli_len !any characters?!
P 0093 6B 30 354 jr eq,rli_6 !none!
P 0095 DE 355 inc rli_len !undo last decrement!
P 0096 26 EF 02 356 sub R15,#2 !backspace & previous!
P 0099 EE 357 inc R14 !reg or ext mem?!
P 009A EA 02 358 djnz R14,rli_7 !exit!
P 009C 8B C2 359 jr rli_4 !reg!
P 009E 36 EE 00 360 rli_7: sbo R14,#0
P 00A1 8B BD 361 jr rli_4

P 00A3 00 ED 363 rli_9: dec rli_len !in case cr!
P 00A5 A6 7C 0D 364 cp TEMP_1,#0D !carriage return?!
P 00A8 6B 03 365 jr z,rT_3 !end input!
P 00AA DE 366 inc rli_len !restore!
P 00AB DA B3 367 rli_2: djnz rli_len,rli_4 !loop for max length!
P 00AD 50 7C 368 rli_3: pop TEMP_1 !original length!
P 00AF 24 ED 7C 369 sub TEMP_1,rli_len !# chars returned!
P 00B2 D8 7C 370 ld rli_len,TEMP_1 !tell caller!
P 00B4 C8 70 371 ld R12,SER_flg_!return read status!
P 00B6 56 70 E3 372 and SER_flg,#NOT (pe LOR bd LOR bo)
P 00B7 373 jr rli_6 !reg for next time!

P 00B9 CF 374 rcf !good return code!
P 00BA 76 EC 9C 375 tm R12,#pe LOR bd LOR bo LOR ad
P 00BD 6B 01 376 jr z,rli_5 !no error!
P 00BF DF 377 sbo !set error return!
P 00C0 50 EF 378 rli_5: pop R15
P 00C2 50 EE 379 pop R14 !original buffer addr!
P 00C4 AF 380 ret

P 00C5 50 ED 382 rli_6: pop rli_len
P 00C7 50 EF 383 pop R15
P 00C9 50 EE 384 pop R14
P 00CB 8B 8D 385 jr ser_read !start over!
P 00CD 386 END ser_rlin

388 GLOBAL
389 ser_rabs PROCEDURE
390 !************************************************************************
391 read absolute
392
393 Purpose = To return input from serial channel
394 of maximum length requested. (Input
395 is not terminated with the receipt of
396 a 'carriage return'. BREAK will
397 terminate read.)
398
399 Note = All other details are as for 'ser rlin'.
400 !************************************************************************
401 ENTRY

P 00CD E6 7E 01 402 ld TEMP_3,#1 !flag => read absolute!
P 00D0 8B 88 403 jr ser_Read
P 00D2 404 END ser_rabs
GLOBAL

PROCEDURE

Purpose = To service IRQ3 by inputting current
character into next available position
in circular buffer.

Input = None.

Output = New character inserted in buffer.
SER_stat , SER_put updated.

Note = 1. If even parity enabled, the software
replaces the eighth data bit with a
parity error flag.

2. If BREAK detection is enabled, and
the received character is null,
the serial input line is monitored to
detect a potential BREAK condition.
BREAK is defined as a zero start bit
followed by 8 zero data bits and a
zero stop bit.

3. If 'buffer full' on entry, 'input
buffer overflow' is flagged.

4. If input echo is on, the character is
immediately sent to the output serial
channel.

5. IMR is modified to allow selected
nested interrupts (see ser init).

ENTRY

ld "SER_tmp1",%03 !read stop bit level!
push imr !save entry IMR!
and imr,"SER_imr" !allow nesting!
push e1
push rp !save user's!
push srp #RAM_STARTr !capture input!
ld rSERchar,SIO !break detect enabled?!
jr rSERcfg,#be !nope.!
jr rSERcfg,#ipl !odd parity enabled?!
jr z,ser_23 !no.
ld rSERtmp2,#80
jr rSERcfg !break detect enabled?!
and rSERtmp2,-03 !set BREAK flag!
lsb marker yet?1
z,ser_24 !not yet!
jmp SERtime !test stop bit!
ld nz,ser_30 !not BREAK!
ret BREAK. Wait for marking!
or rSERflag,#bd !set BREAK flag!
and rSERchar,rSERtmp2 !8 received bits = 0?!
jr ne,ser_30 !no!
rSERtmp1,#1 "test stop bit!"
rSERtmp1,#53 !delay 640 cycles!
wait 1 char time to flush receive shift register!
push SERtime
push PREO x T0!
ret !delay 640 cycles!
P 010A EB F8 470 jr nz,in_loop 1delay (128x10xPREOxT0): 1 1 11 19
471 1 2
472

P 010C 50 6F 473 pop SERtime
P 010E 50 6E 474 pop SERtime 1restore PREO x T0!
P 0110 56 FA F7 475 and IRQ,$LNOT %08 1clear int req!
P 0113 8B 49 476 jr ser_i5 1bye!

P 0115 76 E0 01 477
P 0118 EB 4A 478 jr nZ,ser_i1 1yes,overflow!
P 0121 76 E2 01 479 jr ser_i5 1buffer full?
P 0124 76 EO 01 480 jr ser_i5 1echo on?
P 0127 6E FA E9 481 jr z,ser_i0 1no parity!

P 012A 8B F9 482 ld SERchar,loop 1parity error flag...
P 012D 68 FA 483 ser_i6: tcm IRQ,clear 1bit 7!
P 012F 56 FA EF 484 and IRQ,$LNOT %10 1clear irq bit!

P 0132 56 FA 485 jr ser_i5 1even parity?!
P 0135 76 E2 40 486 ser_i0: tm rSERcfg,leven 1parity?
P 0138 6B 14 487 jr z,ser_i0 1no!

P 013B 6E 488 !calculate parity error flag!

P 013E 8C 07 489 ld rSERTmp1,%7
P 0140 B0 E9 490 clr rSERTmp2 1count 1's here!
P 0142 CO EA 491 ser_20: rrc rSERchar 1bit to carry!
P 0145 8A F9 492 adc rSERTmp2,%0 1update 1's count!

P 0148 7F 493 djnz rSERTmp1,ser_20 1loop till done!

P 014B 8A F9 494 and rSERTmp2,%1 11's count even or odd?
P 014E B2 A9 495 xor rSERchar,rSERTmp2
P 0150 C0 EA 496 rrc rSERchar 1parity error flag...

P 0153 5F 497 rrc rSERchar 1...to bit 7!
P 0156 88 E4 498 ser_22: ld rSERTmph,rSERbufh
P 0159 46 FA 29 499 ld rSERTmph,rSERbufl

P 015C 06 95 500 add rSERTmp1,rSERput 1next char address!
P 015F 8E 501 inc rSERTmph 1in external memory?!

P 0162 8A 1E 502 djnz rSERTmph,ser_12 1yes!
P 0165 FF 9A 503 ld @rSERTmp1,rSERchar 1store char in buf!

P 0168 46 E0 02 504 ser_13: or rSERflg,#bne 1buffer not empty!
P 016B 75 505 rnc rSERput 1update put ptr!
P 016E A2 7E 506 rcr rSERput 1update put ptr!

P 0171 56 E0 02 507 jr nz,ser_i4 1no!

P 0174 B0 E7 508 clr rSERput 1set to start!

P 0177 2A 71 509 ser_i4: op rSERput,rSERget 1if equal, then full!

P 017A 8B 50 510 jr ne,ser_i5 1wrap-around?!

P 017C 55 511 or rSERflg,#bf

P 017F 60 512 ser_i5: ppp rp 1restore user's!

P 0182 6F 513 ld @rSERTmph,rSERchar 1store in buf!

P 0185 50 FB 514 pop imr 1restore entry imr!

P 0188 BF 515 iret

P 018B 46 E0 04 516

P 018E 8B F5 517 jr ser_i1: or rSERflg,#bo 1buffer overflow!

P 0191 69 08 518 jr ser_i5

P 0194 6E 00 519

P 0197 6B 520 ser_i2: adc rSERTmph,#0

P 019A 92 521 lde @rrSERTmph,rSERchar 1store in buf!

P 019D 8B DD 522 jr ser_i3

P 019F 523 42
Purpose = To return one serial input character.

Input = None.

Output = Carry FLAG = 1 if BREAK detected or serial not enabled or buffer overflow = 0 otherwise

TEMP_1 = character

Note = This routine will not return control until a character is available in the input buffer or an error is detected.

ENTRY

push rp

## Program Code

```assembly
P 0170 70 FD push rp
P 0172 31 70 srp #RAM_STARTTr
P 0174 DF scf _in case error!
P 0175 76 E0 8C ser_g1: tm rSERfIg,#sd LOR bd LOR bo
P 0176 54 547 iserial disabled or
P 0177 54 548 BREAK detected or
P 0178 54 549 buffer overflow?

jr nZ,ser

P 0179 54 550 jr nz,ser_g6
P 017A 54 551 jr z,ser_g1
P 017B 54 552 jr z,ser_g1
P 017C 54 553 ld rTEMP_11,rSERbuf1
P 017D 54 554 ld rTEMP_1h,rSERbufh
P 017E 54 555 di _prevent IRQ3 conflict!
P 017F 54 556 add rTEMP_11,rSERget [next char address]

P 0180 54 557 ino rTEMP_1h
P 0181 54 558 djnz rTEMP_1h,ser_g3
P 0182 54 559

P 0183 54 560 ld rTEMP_1,rTEMP_11
P 0184 54 561 jr nz,ser_g6
P 0185 54 562 jr z,ser_g1
P 0186 54 563 op rSERget,rSERlen [wrap-around?]
P 0187 54 564 jr ne,ser_g2
P 0188 54 565 clr rSERget [yes, set to start]
P 0189 54 566 ser_g2: cp rSERget,rSERput [buffer empty if get?]
P 018A 54 567 jr ne,ser_g5
P 018B 54 568 and rSERfIg,#LNOT bne [buffer empty now]
P 018C 54 569 ser_g5: rcf [set good return!]
P 018D 54 570 jr rSERfIg,#LNOT bne [buffer empty now]
P 018E 54 571 ser_g6: pop rp [restore caller's rp!]

P 01A0 AF ret

P 01A1 54 573

P 01A2 54 574 ser_g3: adc rTEMP_1h,#0 [rrTEMP_1 has char addr]
P 01A3 54 575 lde rTEMP_1,#rrTEMP_1 [get char!]
P 01A4 54 576 jr ser_g4 [clean up!]
P 01A5 54 577 END ser_get
```

1-43
LOCAL ser_break PROCEDURE

Purpose = To transmit BREAK on the serial line.
Input = RR14 = break length
Output = None.
Note = BREAK is defined as:
serial out (P37) = 0 for
2 x 28 cycles/loop x RR14 loops
XTAL

RR14 should yield at least 1 bit time
so that the last 'clr SIO' will
have been preceded by at least 1 bit
time of spacing. Therefore, RR14 should
be greater than or equal to
4 x 16 x PREO x TO
        28

ENTRY ser_b1:
clr SIO
decw RR14
jr nz, ser_b1
!wait for last null to be fully transmitted!
jp ser_o1
END ser_break

GLOBAL ser_flush PROCEDURE

Purpose = To flush (clear) the serial input
buffer of characters.
Input = None
Output = Empty input buffer.
Note = This routine might be useful to clear
all past input after a BREAK has been
detected on the line.

ENTRY di
!disable interrupts!
!(to avoid collision with
serial input)!
clr SER_get !buffer start!
clr SER_put != buffer end!
and SER_flg,#%80 !clear status!
ei !re-enable interrupts!
ret
END ser_flush
CONSTANT
wli len := R13
GLOBAL

PROCEDURE
write line

Purpose = To output a character string to serial line, ending with either a 'carriage return' character or the maximum length specified.

Input = RR14 = address of source buffer (in reg/ext memory)
R13 = length

Output = RR14 = updated Carry Flag = 1 if serial not enabled,
= 0 if no error.
R13 = # bytes output (not including auto line feed)

Note = If auto line feed is enabled, a line feed character will be output following each carriage return (ser wlin only).

Entry:

ENTRY
wli 4:
call ser_output
wli len:
call get_src
wli 2:
call ser_output
wli 5:
call get_src
wli len:
call ser_output

iflag => write line!
PROCEDURE

Purpose = To output a character string to serial line for the length specified. (Output is not terminated with the output of a 'carriage return').

Note = All other details are as for 'ser wlin'.

ENTRY

ENTRY

Purpose = To output a given character to the serial line. If the character is a carriage return and auto line feed is enabled, a line feed will be output as well.

Input = R12 = character to output

Note = Equivalent to ser wlin with length = 1.

ENTRY

ENTRY

Purpose = To output a character string to serial line for the length specified. (Output is not terminated with the output of a 'carriage return').

Note = All other details are as for 'ser wlin'.

ENTRY
GLOBAL !for PART II!

ser_output PROCEDURE

Purpose = To output one character to the serial line.

Input = TEMP_1 = character

Output = Carry FLAG = 1 if serial disabled = 0 otherwise.

Note = 1. If even parity is enabled, the eighth data bit is modified prior to character output to SIO.
2. IRQ4 is polled to wait for completion of character transmission before control returns to the calling program.

ENTRY

scf !in case error!
tm SER_flg,#sd !serial disabled?!
jr nz,ser_05 !yes, error!
ser 04: rrc TEMP-1 !count 1's!
adc TEMP-2,#0
dec TEMP-1
jr nz,ser_04 !next bit!
ser 02: ld SIO,TEMP !output character!
ser-01: tcm IRQ,#10- !check IRQ4!
jr nz,ser_01 !wait for complete!
and IRQ,#EF !clear IRQ4!
ret !all ok!

END ser_output

GLOBAL !for PART II!

ser disable PROCEDURE

Purpose = To disable serial I/O operations.

Input = None.

Output = Serial I/O disabled.

ENTRY

di !avoid IRQ3 conflict!

or SER_flg,#ad !serial disabled!
set serial disabled!

and TMR,#FC !disable T0!

and IMR,#$7 !disable IRQ3,4!

and P3M_save,#$BF
P30/7 normal i/o pins!

1d P3M,P3M_save

!re-enable interrupts!

ret

END ser_disable
**Timer/Counter Routines**

```
840 CONSTANT
841 TMP := R13
842 PTR := RR14
843 PTRh := R14
844 GLOBAL

P 0254 PROCEDURE
846 !*****************************************************************************
847 time of day : initialize
848
849 Purpose = To initialize T0 or T1 to function as
850          a time of day clock.
851
852 Input = RR14 = address of parameter list in
853           program memory:
854           1 byte = IMR mask for nestable
855           interrupts
856           1 byte = # of clock ticks per second
857           1 byte = counter # : = $F4 => T0
858           = $F2 => T1
859           1 byte = Counter value
860           1 byte = Prescaler value (unshifted)
861
862 TOD_hr, TOD_min, TOD_sec, TOD_tt
863       initialized to the starting time of
864       hours, minutes, seconds, and ticks
865       respectively.
866
867 Output = Selected timer is loaded and
868       enabled; corresponding interrupt
869       is enabled.
870       R13, R14, R15 modified.
871
872 Note = The cntr and prescaler values provided
873       are those values which will generate an
874       interrupt (tick) the designated # of
875       times per second.
876
877 For example:
878       for XTAL = 8 MHZ, cntr = 250 and
879       prescaler = 40 yield a .01 sec interval;
880       the 2nd byte of the parameter list
881       should = 100 .
882
883 For T0 the instruction at $080C or
884       for T1 the instruction at $080F must
885       result in a jump to the jump table entry
886       for 'tod'.
887
888 The parameter list is not referenced
889       following initialization.
890 !*****************************************************************************
891 ENTRY

P 0254 DC 6C
P 0255 C3 DE
P 0258 C3 DE
P 025A E6 7B 6C
P 025D 8D 02B2'
P 0260

P 0264 DC 8C
P 0265 C3 DE
P 0268 C3 DE
P 026A E6 7B 6C
P 026D 8D 02B2'
P 026F

P 0270 EN END

892  ld   TMP,%TOD_imr
893  ldci  @TMP,%PTR   !imr mask!
894  ldci  @TMP,%PTR   !ticks/second!
895  ld   TEMP,%TOD_imr
896  jp   pres_cnt      !ctr & prescaler!
```

1-48
GLOBAL

PROCEDURE

Interrupt service - time of day

Purpose = To update the time of day clock.

ENTRY

P 0260

push    imr

P 0262 54 FB

and    imr,TOD_imr

P 0265 9F

ei

P 0266 70 FD

push    rp

P 0268 31 60

srp    $RAM_TMR

P 026A 8E

inc    rTODtt

P 026B A2 8D

cp    rTODtt,rTODtic

P 026D EB 13

jr    ne,tod_ex

P 026F B0 E8

clr    rTODtt

P 0271 9E

inc    rTODsec

P 0272 A6 E9 3C

cp    rTODsec,#60

P 0275 EB 0B

jr    ne,tod_ex

P 0277 B0 E9

clr    rTODsec

P 0279 AE

inc    rTODmin

P 027A A6 EA 3C

cp    rTODmin,#60

P 027D EB 03

jr    ne,tod_ex

P 027F B0 EA

clr    rTODmin

P 0281 BE

inc    rTODhr

P 0282 50 FD

tod_ex:  pop    rp

P 0284 8F

di

P 0285 50 FB

pop    imr

P 0287 BF

lret

P 0288

END  tod
GLOBAL

Purpose = To initialize one of the timers

to generate a variable frequency/ variable pulse width output.

Input = RR14 = address of parameter list in program memory:
1 byte = counter @ = %F4 => T0
1 byte = counter @ = %F2 => T1
1 byte = counter value for high interval
1 byte = prescaler (unshifted)

Output = Selected timer is loaded and enabled; corresponding interrupt
is enabled. P36 is enabled as Tout. R13, R14, R15 modified.

Note = The parameter list is not referenced following initialization.

The value of Prescaler x Counter must be > 26 (=%1A) for proper operation.

ENTRY

LD TMP,#PLS_2
ldci @TMP,@PTR !low interval cntr!
ldci @TMP,+PTR !timer addr!
p 0290 80 EE
dew PTR
p 0292 80 EE
dew PRR !back to flag!

ENTRY

lexchange values
xor PLS_1,PLS_2
xor PLS_2,PLS_1
xor PLS_1,PLS_2

ENTRY

ld @PLS_tmr,PLS_1 !load new value!
iret

END pulse_1 -50
GLOBAL

Purpose = To generate an interrupt after a designated amount of time.

Input = RR14 = address of parameter list in program memory:
1 byte = counter 0 : = $F4 => T0
1 byte = Counter value
1 byte = Prescaler value and count mode (to be loaded as is into PREO or PRE1).

Output = Selected timer is loaded and enabled; corresponding interrupt is enabled.
R13, R14, R15 modified.

Note = This routine will initialize the timer for single-pass or continuous mode as determined by bit 0 of byte 3 in the parameter list.
The caller is responsible for providing the interrupt service routine.
The parameter list is not referenced following initialization.

ENTRY

P 02B0

1021 ENTRY

P 02B0

1022 clr TEMP 4

P 02B2

1023 !fall into pre_cnt! -

P 02B2

1024 END delay
Purpose = To get counter and prescaler values from parameter list and modify control registers appropriately.

Input = TEMP_4 = 0 => for 'delay'

Input = TEMP_4 = 1 => for 'pulse'

Input = TOD Imr => for 'tod'

**Assembly complete**
This application note is intended for use by those with either a Z8601 or a Z8611 Microcomputer device. It is assumed that the reader is familiar with both the Z8 and its assembly language, as described in the following documents:

- Z8 Technical Manual (Reset Section) (03-3047-02)
- Z8 Family Z8601, Z8602, Z8603 Product Spec (00-2037-A0)
- Z8 Family Z8611, Z8612, Z8613 Product Spec (00-2038-A0)
- Z8 PLZ/ASM Assembly Language Programming Manual (03-3023-03)

The note briefly discusses the operation of Test Mode, which is a special mode of operation that facilitates testing of both Z8 devices that incorporate an internal program ROM (Z8601, Z8611). There are two problems associated with testing a Z8 with an internal program ROM; the solutions are presented below.

The first problem is: how can the device be tested with standard microprocessor automatic test equipment? To solve this problem, Test Mode causes the Z8 to fetch instructions from Port 1 while it is in the external Address/Data bus mode, instead of fetching instructions from the internal Program ROM. Diagnostic test routines are then forced onto this external bus from the test equipment in the same manner as with microprocessor testing.

The second problem is: since the Test Mode requires that Port 1 operate only in the Address/Data bus mode, how are the other Port 1 modes of operation tested? To solve this problem, an on-chip Test ROM is provided for execution while in Test Mode. The program in the Test ROM checks the other modes of Port 1: input, output, with handshake control, and without handshake control.

Figure 1 compares normal and Test Mode operations in the Z8. (In both normal and Test Mode, program execution begins at address 00CH.)
Test Mode can be entered immediately after reset by driving the RESET input (pin 6) to a voltage of Vcc + 2.5 V. (See the Reset section of the Z8 Technical Manual for a description of the Reset procedure.) Figure 2 shows the voltage waveform needed for Test Mode. After entering Test Mode, instructions are fetched from the internal Test ROM, which is programmed with Port 1 diagnostic routines. The Z8 stays in Test Mode until a normal reset occurs.

The program listing in the ROM is included at the end of this document. Program Listing A (Internal Test ROM Program) is mask programmed into the internal Test ROM of the Z8601. Program Listing B (External Test Program) is an example of a program that could be executed while in Test Mode. It was written as a compliment to the internal Test ROM program, to check the Port input and output functions. To test the other functions of the Z8, the user must execute other programs developed for testing.

The interrupt vectors in the Z8601 Test ROM point to the locations in external memory %800, %803, %806, %809, %80C, %80F. The interrupt vectors in the Z8611 Test ROM point to the locations in external memory %1000, %1003, %1006, %1009, %100C, %100F. This allows the external program to have a 2- or 3-byte jump instruction to each interrupt service routine.

Programs that are run in Test Mode can use an LDE instruction for accessing the Test ROM. The LDC instruction can be used for accessing the program ROM.

The program listing in the ROM is included at the end of this document. Program Listing A (Internal Test ROM Program) is mask programmed into the internal Test ROM of the Z8601. Program Listing B (External Test Program) is an example of a program that could be executed while in Test Mode. It was written as a compliment to the internal Test ROM program, to check the Port input and output functions. To test the other functions of the Z8, the user must execute other programs developed for testing.

The interrupt vectors in the Z8601 Test ROM point to the locations in external memory %800, %803, %806, %809, %80C, %80F. The interrupt vectors in the Z8611 Test ROM point to the locations in external memory %1000, %1003, %1006, %1009, %100C, %100F. This allows the external program to have a 2- or 3-byte jump instruction to each interrupt service routine.

Programs that are run in Test Mode can use an LDE instruction for accessing the Test ROM. The LDC instruction can be used for accessing the program ROM.
Program Listing A. Internal Test ROM Program (continued)

```
P 0029 B9 F7 34 START2: LD P3M R11  \ I START TEST NO H.S.  \!
P 002B 99 F8 35 LD P01M R9  \ I SET P1 TO INPUT  \!
P 002D 1E 36 INC R1  \ I READ & WRITE P1 AS INPUT  \!
P 002E F9 F8 37 LD P01M R15  \ I SET P1 TO OUTPUT  \!
P 0030 1E 38 INC R1  \ I READ & WRITE P1 AS OUTPUT  \!
P 0031 9E E1 39 LD R9 #E1  \ I SAVE RESULTS IN R9  \!
P 0033 C9 F8 40 LD P01M R12  \ I P1&PD=EXT,STK IN,NORMAL  \!
P 0035 8D 0B6D 41 JP VERIFY2  \ I JUMP TO VERIFY #2 ROUTINE  \!
P 0038 42 END TEST  \!
```

Program Listing B. External Test Program

```
47 INTERNAL
48 SETUP

P 0800 49 PROCEDURE ENTRY $ABS $800

P 0800 8D 0800 50
P 0803 8D 0803 51 VECT1: JP VECT1
P 0806 8D 0806 52 VECT2: JP VECT2
P 0809 8D 0809 53 VECT3: JP VECT3
P 080C 8D 080C 54 VECT4: JP VECT4
P 080F 8D 080F 55 VECT5: JP VECT5
P 0812 8F 56 VECT6: JP VECT6

57
P 0813 31 00 58 EXT: DI
P 0815 2C FF 59 SRP #%00
P 0817 3C FF 60 LD R2 #%FF  \ I INITIALIZE P2  \!
P 0819 E6 FF FF 61 LD R3 #%FF  \ I DITTO  \!
P 081C 4C 88 62 LD P2M #%FF  \ I SET P2 TO INPUT  \!
P 081E 5C 00 63 LD R4 #%88  \ I SET P2<>P1 MUX,P3 GRP B MUX  \!
P 0820 9C 86 64 ALSO DUMMY ADDRS HIGH BYTE  \!
P 0822 AC 39 65 LD R5 #%00  \ I DUMMY ADDRS LOW BYTE  \!
P 0824 BC 02 66 LD R9 #%86  \ I P1 OUTPUT MODE VALUE  \!
P 0826 CC 96 67 LD R10 #%39  \ I R10 SETS H.S.,MODE & P2 PULLUPS
P 0828 E6 FF 68 LD R11 #%02  \ I R11 POINTS TO P2 FOR PASS1  \!
P 082A FC 86 69 LD R12 #%96  \ I R12 SETS P01M TO EXT MEM,ETC.
P 082C EC 01 70 LD R13 #%01  \ I R13 POINTS TO P1 FOR PASS1  \!
P 082E E6 10 71 LD R15 #%86  \ I SAME AS R9  \!
P 0830 EC AA 72 LD R14 #%AA  \ I DATA LOADED TO TEST PORT  \!
P 0832 E6 10 10 73 LD #%10  \ I RDY/DAV RESULT PASS 1  \!
P 0835 E6 11 40 74 LD #%11 #%40  \ I DITTO  \!
P 0838 8D 0012 75 DATA LOADED TO TEST PORT
P 0837 76 END SETUP

P 0837 77
P 0831 81 PROCEDURE ENTRY $ABS $831

P 0831 DC 02 82
P 0833 BC 01 83
P 0835 E6 00 84 VERIFY1:LD R13 #%02  \ I R13 POINTS TO P2 FOR PASS2  \!
P 0838 66 E4 50 85 LD R11 #%01  \ I R11 POINTS TO P1 FOR PASS 2  \!
P 083A 6E 66 86 LD P2M #%00  \ I SETS P2 FOR OUTPUT  \!
P 083C 6E 66 87 TCM R4 #%50  \ I FROM HERE TO THERE WE VERIFY  \!
P 083E 6E 66 88 TEST RESULTS FOR I/O WITH H.S.  \!
P 0840 6E 66 89 I BOTH PASS 1&2  \!
```
Program Listing B. External Test Program (continued)

P 083B ED 0880 90 JP NZ FAIL
P 083E 64 10 E5 91 TCM R5 $10
P 0841 ED 0880 92 JP NZ FAIL
P 0844 74 11 E5 93 TM R5 $11
P 0847 ED 0880 94 JP NZ FAIL
P 084A A6 E6 A6 95 CP R6 #$AA
P 084D ED 0880 96 JP NZ FAIL
P 0850 A6 E7 55 97 CP R7 #$55
P 0853 ED 0880 98 JP NZ FAIL
P 0856 66 E8 50 99 TCM R8 #$50
P 0859 ED 0880 100 JP NZ FAIL
P 085C A6 E9 86 101 CP R9 #$86
P 085D E6 10 40 102 LD $10 #$40
P 0862 E6 11 10 103 LD $11 #$10
P 0865 9C 8E 104 LD R9 #$8E
P 0867 6D 0012 105 JP EQ START1
P 086A 8D 0029 106 JP START2
P 086D A6 E9 57 107 VERIFY2:CP R9 #$57
P 0870 6D 0890 109 JP EQ PASS
P 0873 110 END VERIFY
111
112
113 INTERNAL
114 TFAIL
115 PROCEDURE ENTRY $ABS $890
116
117 PASS:JR PASS
118
119 END TFAIL
120
121
122
123 INTERNAL
124 TFAIL
125 PROCEDURE ENTRY $ABS $880
126
127
128 FAIL:JR FAIL
129
130 END TFAIL
131
132 END TESTROM

! IS THIS PASS1?!
! RDY/DAY RESULT PASS 2!
! DITTO!
! P1 IS GOING TO BE AN OUTPUT!
! PASS1 SUCCESSFUL--TRY PASS2!
! PASS2 SUCCESSFUL--TEST NO H.S.
! CHECK RESULT OF I/O NO H.S.TES

IS THIS PASS1?!
I RDY/DAY RESULT PASS 2!
I DITTO!
I P1 IS GOING TO BE AN OUTPUT!
I PASS1 SUCCESSFUL--TRY PASS2!
I PASS2 SUCCESSFUL--TEST NO H.S.
I CHECK RESULT OF I/O NO H.S.TES
Build a Z8-Based Control Computer with BASIC, Part 1

I hope you believe me when I say that I have been waiting years to present this project. For what has seemed an eternity, I have wanted a microcomputer with a specific combination of capabilities. Ideally, it should be inexpensive enough to dedicate to a specific application, intelligent enough to be programmed directly in a high-level language, and efficient enough to be battery operated.

My reason for wanting this is purely selfish. The interfaces I present each month are the result of an overzealous desire to control the world. In lieu of that goal, and more in line with BYTE policy, I satisfy this urge by stringing wires all over my house and computerizing things like my wood stove.

There are many more places I'd like to apply computer monitoring and control. I want to modify my home-security system to use low-cost distributed control rather than central control. I want to try my hand at a little energy management, and, of course, I am still trying to find some reason to install a microcomputer in a car. (How about a talking dashboard?)

Generally, the projects I present each month are designed to be attached to many different commercially available microcomputers through existing I/O (input/output) ports. Most of my projects are applicable for use on the small (by IBM standards) computers owned by many readers, but, unfortunately, a typical home-computer system cannot be stuffed under a car seat.

The Z8-BASIC Microcomputer is a milestone in low-cost microcomputer capability.

The time has come to present a versatile “Circuit Cellar Controller” board for some of these more ambitious control projects. I decided not to adapt an existing single-board computer, which would be larger, more expensive, and generally limited to machine-language programming. Instead, I started from scratch and built exactly what I wanted.

The microcomputer/controller I developed is called the Z8-BASIC Microcomputer. Its design and application will be presented in a two-part article beginning this month. In my opinion, it is a milestone in low-cost microcomputer capability. It can be utilized as an inexpensive tiny-BASIC computer for a variety of changing applications, or it can be dedicated to specialized tasks, such as security control, energy management, solar-heating-system monitoring, or intelligent-peripheral control.

[Editor's Note: We are using the term “tiny BASIC” generically to denote a small, limited BASIC interpreter. The term has been used to refer to some specific commercially available products based on the Tiny BASIC concept promulgated by the People's Computer Company in 1975....RSS] The entire computer is slightly larger than a 3 by 5 file card, yet it includes a tiny-BASIC interpreter, 4 K bytes of program memory, one RS-232C serial port and two parallel I/O ports, plus a variety of other features. (A condensed functional specification is shown in the “At a Glance” text box.) Using a Zilog Z8 microcomputer integrated circuit and Z6132 4 K byte 8-bit read/write memory device, the Z8-BASIC Microcomputer circuit board is completely self-contained and optimized for use as a dedicated controller.

To program it for a dedicated application, you merely attach a user terminal to the DB-25 RS-232C connector, turn the system on, and type in a BASIC program using keywords such as GOTO, IF, GOSUB, and LET. Execution of the program is started by typing RUN. If you need higher speed than BASIC provides, or if you just want to experiment with the Z8 instruction set, you can use the
Once the application program has been written and tested with the aid of the terminal, the finished program can be transferred to an EPROM (erasable programmable read-only memory) via a memory-dump program and the terminal disconnected. Next, the Z8671 memory component is removed from its socket and either a type-2716 (2 K by 8-bit) or type-2732 (4 K by 8-bit) EPROM is plugged into the lower 24 pins. (The choice of EPROM depends upon the length of the program.) When the Z8 board is powered up, the stored program is immediately executed. The EPROM devices and the Z6132 read/write memory device are pin-compatible. Permanent program storage is simply a matter of plugging an EPROM into the Z6132’s socket.

There is much more power on this board than is alluded to in this simple description. That is why I decided to use a two-part article to explain it. This month, I’ll discuss the design of the system and the attributes of the Z8 and Z6132. Next month, I’ll describe external interfacing techniques, a few applications, and the steps involved in transferring a program into an EPROM.

Single-Chip Microcomputers

The central component in the Z8-BASIC Microcomputer is a member of the Zilog Z8 family of devices. The specific component used, the Z8671, is just one of them. Unlike a microprocessor, such as the well-known Zilog Z80, the Z8 is a single-chip microcomputer. It contains programmable (read/write) memory, read-only memory, and I/O-control circuits, as well as circuits to perform standard processor functions. Microprocessors such as the Z80 or the Intel 8080 require support circuitry to make a functional computer system. A single-chip microcomputer, on the other hand, can function solely on its own.

The concept is not new. Single-chip microcomputers have been around for quite a while, and millions of them are used in electronic games. The designers of the Z8, however, raised the capabilities of single-chip microcomputers to new heights and provided many powerful features usually found only in general-application microprocessors.

Typically, single-chip microcomputers have been designed for intensive applications. Under program control, the Z8 can be configured as a stand-alone microcomputer using 2 K to 4 K bytes of internal ROM, as a traditional microprocessor with as much as 120 K to 124 K bytes of external memory, or as a parallel-processing unit working with other computers. The Z8 could be used as a controller in a microwave oven or as the processor in a stand-alone data-entry terminal complete with floppy-disk drives.

Getting Specific: The Z8671

The member of the Z8 family used in this project is the Z8671. This component differs from the garden-variety Z8601 chiefly in the contents of the ROM set at the factory. The pinout specification of the Z8671 is shown in figure 1b, and the package is shown in photo 2 on page 41. The Z8671 package contains the processor circuitry, 2 K bytes of ROM (preprogrammed with a tiny-BASIC interpreter and a debugging monitor), 32 I/O lines, and 144 bytes of programmable (read/write) memory.

The operational arrangement of memory-address space is shown in figure 1c. The internal read/write memory is actually a register file (illustrated in figure 2) composed of 124 general-purpose registers (R4 thru R127), 16 status-control registers (R240 thru R255), and 4 I/O-port registers (R0 thru R3). Any general-purpose register can be used as an accumulator, address pointer, index register, or as part of the internal stack area. The significance of these registers will be explained when I describe the tiny-BASIC/Debug interpreter/monitor.

The 32 I/O lines are grouped into four separate ports and treated internally as 4 registers. They can be configured by software for either input or output and are compatible with

Photo 1: A prototype of the versatile "Circuit Cellar Controller," formally called the Z8-BASIC Microcomputer. The printed-circuit board measures 4 by 4½ inches and has a 44-pin (two-sided 22-pin) edge connector with contacts on 0.156-inch centers. A 2716 or 2732 EPROM can be substituted for the Z6132 Quasi-Static memory, plugging into the same socket.
Figure 1a: Block diagram of the Zilog Z8-family single-chip microcomputers. Their architecture allows these devices to serve in either memory- or I/O-intensive applications. This figure and figures 1b, 1c, 2, 3, and 4 were provided through the courtesy of Zilog Inc.

The Z8 has forty-seven instructions, nine addressing modes, and six interrupts. Using a 7.3728 MHz crystal (producing a system clock rate of 3.6864 MHz) most instructions take about 1.5 to 2.5 μs to execute. Ordinarily, you would not be concerned about single-chip-microcomputer instruction sets and interrupt handling because the programs are mask-programmed into the ROM at the factory. In the Z8671, however, only the BASIC/Debug interpreter is preprogrammed. Using this interpreter, you can write machine-language programs that can be executed through subroutine calls written in BASIC. This feature greatly enhances the capabilities of this tiny computer and potentially allows the software to control high-speed peripheral devices. (A complete discussion of the Z8 instruction set and interrupt structure is beyond the scope of this article. The documentation accompanying the Z8-BASIC Microcomputer Board describes the instruction set in detail.)

The final area of concern is communication. The Z8 contains a full-duplex UART (universal asynchronous receiver/transmitter) and two counter/timers with prescalers. One of the counters divides the 7.3728 MHz crystal frequency to one of eight standard data rates. With the Z8671, these rates range between 110 and 9600 bps (bits per second) and are switch- or software-selectable.

A block diagram of the serial-I/O section is shown in figure 3. Serial data is received through bit 0 of port 3 and transmitted from bit 7 of port 3. While the Z8 can be set to transmit odd parity, the Z8671 is preset for 1 start bit, 8 data bits, no parity, and 2 stop bits. Received data must have 1 start bit, 8 data bits, at least 1 stop bit, and no parity (in this configuration).

Quasi-Static Memory
A limiting factor in small controller
designs has always been the trade-off between memory size and power consumption. To keep the number of components down and simplify construction, a designer generally selects a limited quantity of static memory. Frequently, the choice is to use two type-2114 1 K by 4 NMOS (negative-channel metal-oxide semiconductor) static-memory devices. In practice, however, the 1 K-byte memory size thereby provided is rather limited. It would be much better to expand this to at least 4 K bytes. Unfortunately, eight 2114 chips require considerably more circuit-board space and consume about 0.7 amps at +5 V. Not only would this make the design ill suited for battery power, it could never fit on my 4-by-4½-inch circuit board.

Another approach is to use dynamic memory, as in larger computers. Dynamic memory costs less, bit for bit, than static memory and consumes little power. Unfortunately, most dynamic-memory components require three separate operating voltages and special refresh circuitry. Adding 4 K bytes of dynamic memory would probably take about twelve chips. The advantages gained in reduced power consumption hardly justify the expense and effort.

The solution to this problem, surprisingly enough, also comes from Zilog, in the form of the Z6132 Quasi-Static Memory. The Z6132, shown in photo 4 on page 43, is a 32 K-bit dynamic-memory device, organized into 4 K 8-bit (byte-size) words. It uses single-transistor dynamic bit-storage cells, but the device performs and controls its own data-refresh operations in a manner that is completely invisible to the user and the rest of the system. This eliminates the need for external refresh circuitry. Also, the Z6132 requires only a +5 V power supply. The result is a combination of the design convenience of static memory and the low power consumption of dynamic memory. All 4 K bytes of memory fit in a single 28-pin dual-inline package, which typically draws about 30 milliamps.

An additional benefit in using the Z6132 is that it is pin-compatible with standard type-2716 (2 K by 8-bit) and type-2732 (4 K by 8-bit) EPROMs. This feature is extremely beneficial when you are configuring this Z8 board for use as a dedicated controller. As previously mentioned, the Z6132 can be removed and an EPROM inserted in the low-order 24 pins of the same socket. Thus, any program written and operating in the Z6132 memory can be placed in a nonvolatile EPROM. (There are some program written and operating in the Z6132 memory can be placed in a nonvolatile EPROM. (There are some

Figure 1c: The operational arrangement of memory-address space in the Z8 family. The regions labeled “program memory” and “data memory” may map to the same physical memory, or two separate banks may be used, selected through one bit of I/O port 3. The internal programmable (read/write) memory is a register file containing 124 general-purpose registers, 16 status-control registers, and 4 I/O-port registers.

Figure 2: An expanded view of the register-memory section of figure 1c, showing the organization of the register file. Any general-purpose register can be used as an accumulator, address pointer, index register, or as part of the internal stack area.

Photo 2: The Zilog Z8671 single-chip microcomputer, a member of the Z8 family of devices. This dual-line package contains the processor circuitry, 2 K bytes of ROM, 32 I/O lines, and 144 bytes of programmable memory.
Photo 3: A photomicrograph of the silicon chip containing the working parts of a Z8 microcomputer.

<table>
<thead>
<tr>
<th>Z8-BASIC Microcomputer</th>
<th>Documentation includes:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Z6132 Product Specification</td>
</tr>
<tr>
<td></td>
<td>BASIC/Debug Manual</td>
</tr>
<tr>
<td></td>
<td>Z8-BASIC Microcomputer Construction/Operator’s Manual</td>
</tr>
<tr>
<td>Assembled and tested...</td>
<td>$170</td>
</tr>
<tr>
<td>Kit...</td>
<td>$140</td>
</tr>
</tbody>
</table>

Z8-BASIC Microcomputer power supply
(Size: 2% by 4% inches)
Provides:
+5 V, 300 mA
+12 V, 50 mA
-12 V, 50 mA
Assembled and tested...$35
Kit... $27

The following items are available from:
The MicroMint Inc
917 Midway
Woodmere NY 11598
Telephone.
(800) 645-3479 (for orders)
(516) 374-6793 (for technical information)

All printed-circuit boards are solder-masked and silk-screened.
The documentation supplied with the Z8 board includes approximately 200 pages of materials. It is available separately for $25. This charge will be credited toward any subsequent purchase of the Z8 board.
Please include $4 for shipping and handling. New York residents please include 7% sales tax.
At a Glance

Name
Z8-BASIC Microcomputer

Processor
Zilog Z8-family Z8671 8-bit microcomputer with programmable (read/write) memory, read-only memory, and I/O in a single package. The Z8671 includes a 2 K-byte Tiny-BASIC/Debug resident interpreter in ROM, 144 bytes of scratchpad memory, and 32 I/O lines. System uses 7.3728 MHz crystal to establish clock rate. Two internal and four external interrupts.

Memory
Uses Z6132 4 K-byte Quasi-Static Memory (pin-compatible with 2716 and 2732 EPROMs); 2 K-byte ROM in Z8671. Memory externally expandable to 62 K bytes of program memory and 62 K bytes of data memory.

Input/Output
Serial port: RS-232C-compatible and switch-selectable to 110, 150, 300, 1200, 2400, 4800, and 9600 bps. Parallel I/O: two parallel ports; one dedicated to input, the other bit-programmable as input or output; programmable interrupt and handshaking lines; LSTTL-compatible. External I/O: 16-bit address and 8-bit bidirectional data bus brought out to expansion connector.

BASIC Keywords
GOTO, GO@, USR, GOSUB, IF...THEN, INPUT, LET, LIST, NEW, REM, RETURN, RUN, STOP, IN, PRINT, PRINT HEX. Integer arithmetic/logic/operators: +, -, /, "and AND; BASIC can call machine-language subroutines for increased execution speed; allows complete memory and register interrogation and modification.

Power-Supply Requirements
+5 V ±5% at 250 mA
+12 V ±10% at 30 mA
−12 V ±10% at 30 mA
(The 12 V supplies are required only for RS-232C operation.)

Dimensions and Connections
4- by 4½-inch board; dual 22-pin (0.156-inch) edge connector. 25-pin RS-232C female D-subminiature (DB-25S) connector; 4-pole DIP-switch data-rate selector.

Operating Conditions
Temperature: 0 to 50°C (32 to 122°F)
Humidity: 10 to 90% relative humidity (noncondensing)
Figure 3: Block diagram of the serial-I/O section of the Z8-family microcomputers. The Z8 contains a full-duplex UART (universal asynchronous receiver/transmitter). The data rates are derived from the clock-rate crystal frequency. Serial data is received through bit 0 of port 3 and is transmitted from bit 7 of port 3. An interrupt is generated within the Z8 whenever transmission or reception of a character has been completed.

Z8-BASIC Microcomputer

Figure 5 on pages 46 and 47 is the schematic diagram of the seven-integrated-circuit Z8-BASIC Microcomputer Board, shown in prototype form, with a power supply, in photo 5. IC1 is the Z8671 microcomputer, the member of the Z8 family that contains Zilog’s 2 K-byte BASIC/Debug software in read-only memory. IC2 is the Z6132 Quasi-Static Memory, and IC3 is an 8-bit address latch. Under ordinary circumstances, the Z6132 is capable of latching its address internally, but IC3 is included to allow EPROM operation. IC4 and IC5 form a hard-wired memory-mapped input port used to read the data-rate-selection switches. IC6 and IC7 provide proper voltage-level conversion for RS-232C serial communication.

The seven-integrated-circuit computer typically takes about 200 milliamps at +5 V. The +12 V and -12 V supplies are required only for operating the RS-232C interface. Power required is typically about 25 milliamps on each.

The easiest way to check out the Z8-BASIC Microcomputer after assembly is to attach a user terminal to the RS-232C connector (J2) and set the data-rate-selector switches to a convenient rate. I generally select 1200 bps, with SW2 closed and SW1, SW3, and SW4 open. After applying power, simply press the RESET push button.

Pressing RESET starts the Z8’s initialization procedure. The program reads location hexadecimal FFFD in memory-address space, to which the data-rate-selector switches are wired to respond. When it has acquired this information, it sets the appropriate data rate and transmits a colon to the terminal. At this point, the Z8 board is completely operational and programs can be entered in tiny BASIC.

Photo 6: The Z8-BASIC Microcomputer in operation, communicating with a video terminal (here, a Digital Equipment Corporation VT8E). A memory-dump routine, written using the BASIC/Debug interpreter, is shown on the display screen. The starting address of the dump is the beginning of the user-memory area; the hexadecimal values displayed are the ASCII (American Standard Code for Information Interchange) values of the characters that make up the first line of the memory-dump program.
Figure 4: Block diagram of the Zilog Z6132 Quasi-Static Memory component. This innovative part stores 32 K bits in the form of 4 K bytes, using single-transistor dynamic random-access bit-storage cells, but all refresh operations are controlled internally. The memory-refresh operation is completely invisible to the user and the other components in the system. The Z6132 draws about 30 milliamps from a single +5 V power supply.

(With the simple address selection employed in this circuit, the data-rate switches will be read by an access to any location in the range hexadecimal C000 thru FFFF. This should not unduly restrict the versatility of the system in the type of application for which it was designed.)

BASIC/Debug Monitor
I’ll go into the features of the tiny BASIC interpreter in greater detail next month, but I’m sure you are curious about the capabilities present in a 2 K-byte BASIC system.

Essentially an integer-math dialect of BASIC, Zilog’s BASIC/Debug software is specifically designed for process control. It allows examination and modification of any memory location, I/O port, or register. The interpreter processes data in both decimal and hexadecimal radices and accesses machine-language code as either a subroutine or a user-defined function.

BASIC/Debug recognizes sixteen keywords: GOTO, GO@, USR, GOSUB, IF...THEN, INPUT, IN, LET, LIST, NEW, REM, RUN, RETURN, STOP, PRINT, and PRINT HEX. Standard syntax and mathematical operators are used.

The Z8 board is not my idea of what should be available; it is available now.

Twenty-six numeric variables, designated by the letters A thru Z, are supported. Variables can be used to designate program line numbers. For example, GOSUB B·100 and GOTO A·B·C are valid expressions.

In my opinion, the 2 K-byte interpreter is extremely powerful. Because it operates easily on register and memory locations, arrays and blocks of data can be easily manipulated.

(Full appreciation of the Z8-BASIC Microcomputer comes after a complete review of the operating manuals and a little experience. Documentation approximately 200 pages long is supplied with the unit; the documentation is also available separately.)

In Conclusion
It’s easy to get spoiled using a large computer as a simple control device. I have heard of many inexpensive interfaces that, when attached to any computer, supposedly perform control and monitoring miracles. Frequently overlooked, however, is the fact that implementation of these interfaces often requires the software-development tools and hardware-interfacing facilities of relatively large systems. The Z8-BASIC Microcomputer, with its interpretive language, virtually eliminates the need for costly development systems with memory-consuming text editors, assemblers, and debugging programs.
If you need a proportional motor-speed control for your solar-heating system, you don't have to dedicate your Apple II or shut off your heating system when you balance your checkbook. From now on, there is a small, cost-effective microcomputer specifically designed for such applications. The Z8 board described in this article is not my idea of what should be available; it is available now.

Next Month:
I will elaborate on interfacing and applications for the Z8-BASIC Microcomputer.

Acknowledgment
Special thanks to Steve Walters and Peter Brown of Zilog Inc for help in production of this article.

Editor's Note: Steve often refers to previous Circuit Cellar articles as reference material for the articles he presents each month. These articles are available in reprint books from BYTE Books, 70 Main St, Peterborough NH 03458. Ciarcia's Circuit Cellar covers articles appearing in BYTE from September 1977 thru November 1978. Ciarcia's Circuit Cellar, Volume II presents articles from December 1978 thru June 1980.

Figure 5: Schematic diagram of the Circuit Cellar Z8-BASIC Microcomputer. Five jumper connections are provided so different memory devices can be used. For general-purpose use and program development, the 4 K-byte Z6132 read/write memory device will be used; for dedicated applications, two kinds of EPROMs can be substituted in the same integrated-circuit socket. Standard 450 ns type-2716 or type-2732 EPROM chips can be used. The connection labeled "32 K" should be closed if a type-2732 EPROM is installed; the connection labeled "16 K" should be closed for use of a type-2716 EPROM.

The pull-up resistors adjacent to IC4 (the 74LS244 buffer) are contained in a SIP (single-inline package).
The Z8-BASIC Microcomputer system described in this two-part article is unlike any computer presently available for dedicated control applications. Based on a single-chip Zilog Z8 microcomputer with an onboard tiny-BASIC interpreter, this unit offers an extraordinary amount of power in a very small package. It is no longer necessary to use expensive program-development systems. Computer control can now be applied to many areas where it was not previously cost-effective.

The Z8-BASIC Microcomputer is intended for use as an intelligent controller, easy to program and inexpensive enough to dedicate to specific control tasks. It can also serve as a low-cost tiny-BASIC computer for general interest. Technical specifications for the unit are shown in the "At a Glance" box.

Last month I described the design of the Z8-BASIC Microcomputer hardware and the architectures of the Z8671 microcomputer component and Z6132 32 K-bit Quasi-Static Memory. This month I'd like to continue the description of the tiny-BASIC interpreter, discuss how the BASIC program is stored in memory, and demonstrate a few simple applications.

Process-Control BASIC

The BASIC interpreter contained in ROM (read-only memory) within the Z8671 is officially called the Zilog BASIC/Debug monitor. It is essentially a 2 K-byte integer BASIC which has been optimized for speed and flexibility in process-control applications.

There are 15 keywords: GOTO, GO@, USR, GOSUB, IF...THEN, INPUT, IN, LET, LIST, NEW, REM, RUN, RETURN, STOP, PRINT (and PRINT HEX). Twenty-six numeric variables (A through Z) are supported; and numbers can be ex-
pressed in either decimal or hexadecimal format. BASIC/Debug can directly address the Z8's internal registers and all external memory. Byte references, which use the "@" character followed by an address, may be used to modify a single register in the processor, an I/O port, or a memory location. For example, @4096 specifies decimal memory location 4096, and @%F6 specifies the port-2 mode-control register at decimal location 246. (The percent symbol indicates that the characters following it are to be interpreted as a hexadecimal numeral.) To place the value 45 in memory location 4096, the command is simply, @4096=45 (or @%1000= %2D).

Command abbreviations are standard with most tiny-BASIC interpreters, but this interpreter allows some extremes if you want to limit program space. For example:

IF 1>X THEN GOTO 1000
   can be abbreviated
IF 1>X 1000
   PRINT "THE VALUE IS ";S

One important difference between most versions of BASIC and Zilog's BASIC/Debug is that the latter allows variables to contain statement numbers for branching, and variable storage is not cleared before a program is run. Statements such as GOSUB X or GOTO A+E-Z are valid. It is also possible to pass values from one program to another. These variations serve to extend the capabilities of BASIC/Debug.

In my opinion, the main feature that separates this BASIC from others is the extent of documentation supplied with the Z8671. Frequently, a computer user will ask me how he can obtain the source-code listing for the BASIC interpreter he is using. Most often, I have to reply that it is not available. Software manufacturers that have invested many man-years

---

**At a Glance**

**Name**
Z8-BASIC Microcomputer

**Processor**
Zilog Z8-family Z8671 8-bit microcomputer with programmable (read/write) memory, read-only memory, and I/O in a single package. The Z8671 includes a 2 K-byte tiny-BASIC/Debug resident interpreter in ROM, 144 internal 8-bit registers, and 32 I/O lines. System uses 7.3728 MHz crystal to establish clock rate. Two internal and four external interrupts.

**Memory**
Uses Z8132 4 K-byte Quasi-Static Memory (pin-compatible with 2716 and 2732 EPROMs); 2 K-byte ROM in Z8671. Memory externally expandable to 62 K bytes of program memory and 62 K bytes of data memory.

**Input/Output**
Serial port: RS-232C-compatible and switch-selectable to 110, 150, 300, 1200, 2400, 4800, and 9600 bps. Parallel I/O: two parallel ports; one dedicated to input, the other bit-programmable as input or output; programmable interrupt and handshaking lines; LSTTL-compatible. External I/O: 16-bit address and 8-bit bidirectional data bus brought out to expansion connector.

**BASIC Keywords**
GOTO, GO@, USR, GOSUB, IF...THEN, INPUT, LET, LIST, NEW, REM, RETURN, RUN, STOP, IN, PRINT, PRINT HEX. Integer arithmetic/logic operators: +, -, /, *, and AND; BASIC can call machine-language subroutines for increased execution speed; allows complete memory and register interrogation and modification.

**Power-Supply Requirements**
+5 V ±5% at 250 mA
+12 V ±10% at 30 mA
−12 V ±10% at 30 mA
(The 12 V supplies are required only for RS-232C operation.)

**Dimensions and Connections**
4- by 4½-inch board; dual 22-pin (0.156-inch) edge connector. 25-pin RS-232C female D-subminiature (DB-25S) connector; 4-pole DIP-switch data-rate selector.

**Operating Conditions**
Temperature: 0 to 50°C (32 to 122°F)
Humidity: 10 to 90% relative humidity (noncondensing)

---

**Photo 2:** The Z8/Micromouth demonstrator. A Z8-BASIC Microcomputer is configured to run a ROM-resident program that exercises the Micromouth speech synthesizer presented in the June Circuit Cellar article. A Micromouth board similar to that shown on the left is mounted inside the enclosure. Six pushbutton switches, connected to a parallel input port on the Z8 board, select various speech-demonstration sequences. The Micromouth board is driven from a second parallel port on the Z8 board.
in a BASIC interpreter are not easily persuaded to give away its secrets.

In most cases, however, a user merely wants to know the location of the GOSUB...RETURN address stack or the format and location of stored program variables. While the source code for BASIC/Debug is also not available (because the object code is mask-programmed into the ROM, you couldn't change it anyway), the locations of all variables, pointers, stacks, etc, are fixed, and their storage formats are defined and described in detail. The 60-page BASIC/Debug user's manual contains this information and is included in the 200 pages of documentation supplied with the Z8-BASIC Microcomputer board. (The documentation is also available separately.)

Memory Allocation

Z8-family microcomputers distinguish between four kinds of memory: internal registers, internal ROM, external ROM, and external read/write memory. (A slightly different distinction can also be made between program memory and data memory, but in this project this distinction is unnecessary.) The register file resides in memory-address space in hexadecimal locations 0 through FF (decimal 0 through 255). The 144 registers include four 1/O- (input/output) port registers, 124 general-purpose registers, and 16 status and control registers. (No registers are implemented in hexadecimal addresses 80 through EF [decimal addresses 128 through 239]).

The 2 K-byte ROM on the Z8671 chip contains the BASIC/Debug interpreter, residing in address space from address 0 to hexadecimal 7FF (decimal 0 to 2047). External memory starts at hexadecimal address 800 (decimal 2048). A memory map of the Z8-BASIC Microcomputer system is shown in figure 1.

When the system is first turned on, BASIC/Debug determines how much external read/write memory is available, initializes memory pointers, and checks for the existence of an auto-start-up program. In a system with external read/write memory, the top page is used for the line buffer, program-variable storage, and the GOSUB...RETURN address stack. Program execution begins at hexadecimal location 800 (decimal 2048). When BASIC/Debug finds no external read/write memory, the internal registers are used to store the variables, line buffer, and GOSUB...RETURN stack. This limits the depth of the stack and the number of variables that can be used simultaneously, but the restriction is not too severe in most control applications. In a system without external memory, automatic program execution begins at hexadecimal location 1020 (decimal 4128).

Program Storage

The program-storage format for BASIC/Debug programs is the same in both types of memory. Each BASIC statement begins with a line number and ends with a delimiter. If you were to connect a video terminal or teletypewriter to the RS-232C serial port and type the following line:

```
100 PRINT "TEST"
```

it would be stored in memory beginning at hexadecimal location 800 as shown in listing 1.

The first 2 bytes of any BASIC statement contain the binary equivalent of the line number (100 decimal equals 64 hexadecimal). Next are bytes containing the ASCII (American Standard Code for Information Interchange) values of characters in the statement, followed by a delimiter byte (containing 00) which indicates the end of the line. The last statement in the program (in this case the only one) is followed by 2 bytes containing the hexadecimal value FFFF, which designates line number 65535.

The multiple-line program in listing 2 further illustrates this storage format.
One final example of this is illustrated in listing 3. Here is a program written to examine itself. Essentially, it is a memory-dump routine which lists the contents of memory in hexadecimal. As shown, the 15-line program takes 355 bytes and occupies hexadecimal locations 800 through 963 (decimal 2048 through 2499). I have dumped the first and last lines of the program to further demonstrate the storage technique.

I have a reason for explaining the internal program format. One of the useful features of this computer is its ability to function with programs residing solely in EPROM. However, the EPROMs must be programmed externally. While I will explain how one with no communication facility.

But if you are willing to spend the time, it is easy to print out the contents of memory and manually load the program into an EPROM. However, it is a memory-dump routine that illustrates how the program is stored in a Microcomputer. The first thing to do is to configure the port-2 and port-3 mode-control registers (hexadecimal F6 and F7, or decimal 246 and 247). Port 2 is bit-programmable. For instance, to configure it for 4 bits input and 4 bits output, you would load F0 into register F6 (246). In this case, I wanted it configured as 8 output bits, so I typed it in the BASIC/Debug command @246=0 (set decimal location 246 to 0).

The data-ready strobe is produced using one of the options on the Z8’s port 3. A Z8 microcomputer has data-available and input-ready handshaking on each of its 4 ports. To set the proper handshaking protocol and use port 2 as I have described, a code of hexadecimal 71 (decimal 113) is placed into the port-2 mode-control register. The BASIC/Debug command is @247=113. The RDY2 and DA lines are connected together to produce the data-available strobe signal.

The result (see photo 2) has three basic functional components. On top of the box is a Z8-BASIC Microcomputer (hereinafter called the "Z8 board") with a 2716 EPROM installed in the memory, a circuit socket, the Z8-board power supply (the wall-plug transformer module is out of view), and six pushbutton switches. Inside the box is a prototype version of the Micromouth speech-processor board (a final-version Micromouth board is shown on the left).

The Micromouth board is jumper-programmed for parallel-port operation (8 parallel bits of data and a data-ready strobe signal) and connected to I/O port 2 on the Z8 board. The Micromouth BUSY line and the BOA 45 53 54 22

The first application I had for the unit was as a demonstration driver for the Micromouth speech-processor board I presented two months ago in the June issue of BYTE. (See "Build a Low-Cost Speech-Synthesizer Interface," in the June 1981 BYTE, page 46, for a description of this project, which uses National Semiconductor's Digitalker chip set.) It's hard to discuss a synthesized-speech interface without demonstrating it, and I didn't want to carry around my big computer system to control the Micromouth board during the demonstration. Instead, I quickly programmed a Z8-BASIC Microcomputer to perform that task. While I was at it, I set six pushbuttons attached to 7 input bits of the Z8 board’s input port mapped into memory-address space at hexadecimal address FFFD (decimal 65533).

The most significant 3 bits of port FFFD are normally reserved for the data-rate-selector switches, but with no serial communication required, the data rate is immaterial and the switches are left in the open position. This makes the 8 bits of port FFFD, which are brought out to the edge connector, available for external inputs. In this case, pressing one of the six pushbuttons selects one of six canned speech sequences.

Coherent sentences are created by properly timing the transmission of word codes to the speech-processor board. This requires nothing more than a single handshaking arrangement and a table-lookup routine (but try it without a computer sometime). The program is shown in listing 4a. The first thing to do is to configure the port-2 and port-3 mode-control registers (hexadecimal F6 and F7, or decimal 246 and 247). Port 2 is bit-programmable. For instance, to configure it for 4 bits input and 4 bits output, you would load F0 into register F6 (246). In this case, I wanted it configured as 8 output bits, so I typed it in the BASIC/Debug command @246=0 (set decimal location 246 to 0).

The data-ready strobe is produced using one of the options on the Z8’s port 3. A Z8 microcomputer has data-available and input-ready handshaking on each of its 4 ports. To set the proper handshaking protocol and use port 2 as I have described, a code of hexadecimal 71 (decimal 113) is placed into the port-2 mode-control register. The BASIC/Debug command is @247=113. The RDY2 and DA lines are connected together to produce the data-available strobe signal.

Lines 1000 through 1030 in listing 4a have nothing to do with demonstrating the Micromouth board. They form a memory-dump routine that illustrates how the program is stored in memory. You notice from the memory dump of listing 4b that the first byte of the program, as stored in the

Listing 1: Simple illustration of BASIC program storage in the Z8-BASIC Microcomputer.

<table>
<thead>
<tr>
<th>100</th>
<th>P</th>
<th>R</th>
<th>I</th>
<th>N</th>
<th>T</th>
<th>200</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>00</td>
<td>64</td>
<td>50</td>
<td>49</td>
<td>48</td>
<td>54</td>
<td>22</td>
</tr>
<tr>
<td>E</td>
<td>S</td>
<td>T</td>
<td>&quot;</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80A</td>
<td>45</td>
<td>53</td>
<td>54</td>
<td>22</td>
<td>00</td>
<td>FF</td>
<td></td>
</tr>
</tbody>
</table>

Listing 2: A multiple-line illustration of BASIC program storage.

100 A=5  
200 B=6  
3005 "A*B=":;A*B

<table>
<thead>
<tr>
<th>100</th>
<th>A</th>
<th>5</th>
<th>200</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>800</td>
<td>00</td>
<td>64</td>
<td>41</td>
<td>3D</td>
</tr>
<tr>
<td>80A</td>
<td>36</td>
<td>00</td>
<td>0B</td>
<td>BD</td>
</tr>
<tr>
<td>814</td>
<td>3B</td>
<td>41</td>
<td>2A</td>
<td>42</td>
</tr>
</tbody>
</table>

Dedicated-Controller Use

The Z8-BASIC Microcomputer can be easily set up for use in intelligent control applications. After being tested and debugged using a terminal, the control program can be written into an EPROM. When power is applied to the microcomputer, execution of the program will begin automatically.
ROM, begins at hexadecimal location 820 (actually at 1020, you remember) rather than 800 as usual. This is to help automatic start-up. The program could actually begin anyplace, but you would have to change the program-pointer registers (registers 8 and 9) to reflect the new address. The 32 bytes between 800 and 820 are reserved for vectored addresses to optional user-supplied I/O drivers and interrupt routines.

**Programming the EPROM**

The first EPROM-based program I ran on the Z8-BASIC Microcomputer was manually loaded. I simply printed out the contents of the Z6132 memory using the program of listing 3 and entered the values by hand into the EPROM programmer. This is fine once or twice, but you certainly wouldn't want to make a habit of it. Fortunately, there are better alternatives if you have the equipment.

Many EPROM programmers are peripheral devices on larger computer systems. In such cases, it is possible to take advantage of the systems' capabilities by downloading the Z8 program directly to the programmer.

The programmer shown in photo 3 is a revised version of the unit I described in a previous article, "Program Your Next ROM in BASIC" (March 1978 BYTE, page 84). It was designed for type-2708 EPROMs, but I have since modified it to program 2716s instead. All I had to do was lengthen the programming pulse to 50 ms and redefine the connections to four pins on the EPROM socket. It still is controlled by a BASIC program and takes less than 2½ minutes to program a type-2716 EPROM device. Refer to the original article for the basic design.

Normally, the LIST function or memory-dump routine cannot be used to transmit data to the EPROM programmer because the listing is filled with extraneous spaces and carriage returns. It is necessary to write a program that transmits the contents of memory without the extra characters required for display formatting. The only data received by the EPROM programmer should be the object code to load into the EPROM.

In writing this program we can take advantage of the Z8's capability of executing machine-language programs directly through the USR and GO@ commands. The serial-input and serial-output subroutines in the BASIC/Debug ROM can be executed independently using these commands. The serial-input driver starts at hexadecimal location 54, and the serial-output driver starts at hexadecimal location 61. Transmitting a single character is simply done by the BASIC statement

```basic
GO@ %61,C
```

where C contains the value to be
A serial character can be transmitted. A serial character can be received by

\[ C = \text{USR} \left( \%54 \right) \]

where the variable \( C \) returns the value of the received data.

To dump the entire contents of the Z6132 memory to the programmer, the statements in listing 5 should be included at the end of your program.

Execution begins when you type GOTO 1000 as an immediate-mode command and ends when all 4 K bytes have been dumped. The transmission rate (110 to 9600 bps) is that selected on the data-rate-selector switches.

Conceivably, this technique could also be used to create a cassette-stor-

Listing 4: A program (listing 4a) that demonstrates the functions of the Micromouth speech synthesizer, operating from a type-2716 EPROM. The simple I/O-address decoding of the Z8 board allows use of the round-figure address of 65000. The program uses a table of vocabulary pointers that has been previously stored in the EPROM by hand. Listing 4b shows a dump of the memory region occupied by the program, proving that storage of the BASIC source code starts at hexadecimal location 820.

\[(4a)\]

\[
\begin{align*}
100 & @246=0 : @247=113 \\
110 & X = @65000 : A = @1400 \\
120 & \text{IF } X = 254 \text{ THEN } @2 = 0 \\
130 & \text{IF } X = 253 \text{ THEN GOTO 500} \\
140 & \text{IF } X = 251 \text{ THEN } A = A + 32 : \text{GOTO 500} \\
150 & \text{IF } X = 247 \text{ THEN } A = A + 64 : \text{GOTO 500} \\
160 & \text{IF } X = 239 \text{ THEN } A = A + 96 : \text{GOTO 500} \\
170 & \text{IF } X = 223 \text{ THEN } A = A + 128 : \text{GOTO 500} \\
180 & \text{IF } X = 222 \text{ THEN } N = 0 : \text{GOTO 300} \\
200 & \text{GOTO 110} \\
300 & @2 = N : N = N + 1 : \text{IF } N = 143 \text{ THEN 110} \\
310 & \text{IF } @65000 < 129 \text{ THEN 310} \\
320 & \text{GOTO 300} \\
500 & @2 = @A : A = A + 1 \\
510 & \text{IF } @65000 < 129 \text{ THEN 510} \\
520 & \text{IF } @A = 255 \text{ THEN GOTO 110} \\
530 & \text{GOTO 500} \\
1000 & Q = 2048 \\
1005 & W = 0 \\
1010 & \text{PRINT HEX(}@Q\text{),}; Q = Q + 1 \\
1015 & W = W + 1 : \text{IF } W = 8 \text{ THEN PRINT" ": GOTO 1005} \\
1020 & \text{IF } Q = 4095 \text{ THEN STOP} \\
1030 & \text{GOTO 1010} \\
\end{align*}
\]

\[(4b)\]

\[
\text{:goto 1000}
\]

<table>
<thead>
<tr>
<th>FF</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
<th>FF</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
</tr>
<tr>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
<td>FF</td>
</tr>
<tr>
<td>0</td>
<td>64</td>
<td>40</td>
<td>32</td>
<td>34</td>
<td>36</td>
<td>3D</td>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3A</td>
<td>40</td>
<td>32</td>
<td>34</td>
<td>37</td>
<td>3D</td>
<td>31</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>0</td>
<td>0</td>
<td>6E</td>
<td>58</td>
<td>3D</td>
<td>40</td>
<td>36</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>20</td>
<td>3A</td>
<td>41</td>
<td>3D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>31</td>
<td>34</td>
<td>30</td>
<td>30</td>
<td>0</td>
<td>0</td>
<td>78</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>49</td>
<td>46</td>
<td>20</td>
<td>58</td>
<td>3D</td>
<td>32</td>
<td>35</td>
<td>34</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>54</td>
<td>48</td>
<td>45</td>
<td>4E</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0! AT 1015
Listing 5: BASIC statements that print out the entire contents of the 4 K bytes of user memory, for use with a communicating EPROM programmer.

1000 X = %800 : REM BEGINNING OF USER MEMORY
1010 GO@ %61,X : REM TRANSMIT CONTENTS OF LOCATION X
1020 X = X + 1 : IF X = %1801 THEN STOP
1030 GOTO 1010

Listing 6: A simple BASIC program segment to demonstrate the concept of the "black box" method of modifying data being transmitted through the Z8-BASIC Microcomputer.

100 @246 = 0 : @247 = 113 : REM SET PORT 2 TO BE OUTPUT
110 @2 = X : REM X EQUALS THE DATA TO BE TRANSMITTED through BF07 (decimal 48896 through 48903). When the Z8671 performs an output operation to the channel address, the channel is initialized for acquiring data, while data is read from the channel when the Z8671 performs an input operation on the channel's address.

Intelligent Communication

Another possible use for the Z8-BASIC Microcomputer is as an intelligent "black box" for performing predetermined modification on data being transmitted over a serial communication line. The black box has two DB-25 RS-232C connectors, one for receiving data and the other for retransmitting it. The intelligence of the Z8-BASIC Microcomputer, acting as the black box, can perform practically any type of filtering, condensing, or translating of the data going through.

Perhaps you have an application where continuous raw data is transmitted, but you would rather just keep a running average or flag deviations from preset limits at the central monitoring point rather than contend with everything. The Z8 board can be programmed to digest all the raw data coming down the line and pass on only what's pertinent. Another such black-box application is to use the Z8 board as a printer buffer. Photo 4 shows the interface hardware of one specific application, connected to port 2, any program data sent to register 2 will be transmitted serially at the data rate selected on the four-position DIP switch (between 50 to 19200 bps). The Z8 board, configured with two serial ports, is used to process raw data moving through it. Data is received on one side, digested, and retransmitted in some more meaningful form from the other port. Such a configuration could also be used to connect two peripheral devices that have radically different data rates.
which I used to attach a high-speed computer to a very slow printer. The host computer transmitted data to the Z8 board at 4800 bps. Since the receiving serial port used had to be bidirectional to handshake with the host computer, I added another serial output to the Z8 board for transmitting characters to the printer. Only three integrated circuits were required to add a serial output port. A schematic diagram is shown in figure 3. The UART (universal asynchronous receiver/transmitter, shown as IC1) is driven directly from port 2 on the Z8 board (port 2 could also be used to directly drive a parallel-interface printer), and IC2 supplies the clock signal for the desired data rate. Of course, the UART could have been attached to the data and address buses directly, but this was easier.

Transmitting a character out of this serial port requires setting the port-2 and port-3 mode-control registers as before. After that, any character sent to port 2 will be serially transmitted.

<table>
<thead>
<tr>
<th>Number</th>
<th>Type</th>
<th>5V</th>
<th>GND</th>
<th>12V</th>
</tr>
</thead>
<tbody>
<tr>
<td>IC1</td>
<td>74LS04</td>
<td>14</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>IC2</td>
<td>74LS30</td>
<td>14</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>IC3</td>
<td>74LS02</td>
<td>14</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>IC4</td>
<td>74LS373</td>
<td>20</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>IC5</td>
<td>ADC0808</td>
<td>see schematic diagram</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IC6</td>
<td>LM301</td>
<td>4</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>IC7</td>
<td>74LS00</td>
<td>14</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2: Schematic diagram of an A/D converter. This 8-bit, eight-channel unit has a unipolar input range of 0 to +5 V, with the eight output channels addressed as I/O ports mapped into memory-address space at hexadecimal addresses BF00 through BF07.
Figure 3: Schematic diagram of an RS-232C serial output port for the "black box" communication application of the Z8-BASIC Microcomputer. The Z8671 must be configured by software to provide the proper signals: one such signal, DAV2, is derived from two bits of I/O port 3 on the Z8671. The pin numbers shown in the schematic diagram for P3, and P3a are pins on the Z8671 device itself, not pins or sections on the card-edge connector, as are P2a through P2.

The minimum program to perform this is shown in listing 6. This circuit can also be used for downloading programs to the EPROM programmer.

In Conclusion
It is impossible to describe the full potential of the Z8-BASIC Microcomputer in so few pages. For this reason, considerable effort has been taken to fully document its characteristics. I have merely tried to given an introduction here.

I intend to use the Z8-BASIC Microcomputer in future projects. I am interested in any applications you might have, so let me know about them, and we can gain experience together.

Special thanks to Steve Walters and Peter Brown of Zilog Inc for their aid in producing these articles.

BASIC/Debug is a trademark of Zilog Inc.
INTRODUCTION

The Z8601 is a single-chip microcomputer with four 8-bit I/O ports, two counter/timers with associated prescalers, asynchronous serial communication interface with programmable baud rates, and sophisticated interrupt facilities. The Z8601 can access data in three memory spaces: 2K bytes of on-chip ROM and 62K bytes of external program memory, 144 bytes of on-chip Register, and 62K bytes of external data memory.

The Z8671 is a Z8601 with a Basic/Debug Interpreter and Debug monitor preprogrammed into the 2K bytes of on-chip ROM. This application note discusses some considerations in designing a low-complexity board that runs the Basic/Debug Interpreter and Debug monitor with an external 4K bytes of RAM and 2K bytes of ROM. The board stands alone, allowing users to connect it with a terminal via an RS232 connector and run the Basic/Debug Interpreter.

The user of this board can run Basic/Debug with little knowledge of the Z8601. The board, however, derives its power through its ability to execute assembly language programs. To use the board to its full potential, the Z8 Technical Manual (document #03-3047-02) and the Z8 PLZ/ASM Manual (document #03-3023-03) should be read. The Z8 Basic/Debug Software Reference Manual (document #03-3134-00) provides general information, statement syntax, memory allocations, and other material regarding Basic/Debug and the Debug monitor provided by the Z8671. There are also two documents describing the Z8671; these are the Z6132 Product Specification (document #00-2028-A), and the Interfacing to the Z6132 Intelligent Memory Application Note (document #00-2102-A).

Basic/Debug

Basic/Debug is a subset of Dartmouth Basic, which interprets Basic statements and executes assembly language programs located in memory. Basic/Debug can implement all the Dartmouth Basic commands directly or indirectly.

One advantage to programming in Basic/Debug is the interactive programming approach realized because Basic/Debug is interpreted, not assembled or compiled. Modules are tested and debugged using the interactive monitor provided with Basic/Debug. Using Basic/Debug saves program development time by providing higher-level language statements that simplify program development. Using the INPUT and PRINT statements simplify debugging.

The Z8671 Microcomputer

Basic/Debug controls the memory interface, serial port, and other housekeeping functions performed by the assembly language programmer.

The Z8671 uses ports 0 and 1 for communicating with external memory. Port 1 provides the multiplexed address/data lines (AD0-AD7); port 0 supplies the upper address bits (A8-A15). The Z8671 also uses the serial communications port for communicating with a terminal. Serial communication takes two pins from port 3, leaving six I/O pins from port 3 available to the user. The serial communication interface uses one of the two counter/timers on the Z8671 chip.

All other functions and features on the Z8601 are available with the Z8671. The user may reconfigure the Z8671 in software as a Z8601 if desired.

Applying the Z8671

Applications of the Z8671 range from a low-complexity home microcomputer that is memory intensive to an inexpensive, I/O-oriented microcontroller.

For home computer users, Basic/Debug is used like other available Basic interpreters. The Z8671, however, has many advantages over other computers. For example, the programmer can use the available functions such as interrupts to perform sophisticated tasks that are beyond the scope of other computer products. There is also a counter/timer
that is used as a watchdog counter, a time-of-day clock, a variable pulse width generator, a pulse width measurement device, and a random number generator.

As an inexpensive microcontroller, Basic/Debug speeds program development time by calling assembly language subroutines (for time critical applications) and by supplying high-level Basic language statements that simplify the programming of noncritical subroutines.

ARCHITECTURE

Two major design goals were set for this Z8671 Basic board. First, the board was to be simple. Second, the board needed to allow the user to write Basic programs and to utilize the features of the Z8601.

Overview

The board has seven IC packages:

- Z8671 (Z8601 preprogrammed with Basic/Debug)
- Z6132 (4K bytes of pseudo-static RAM)
- 2716 (2K bytes of EPROM)
- 1488 (RS232 line driver)
- 1489 (RS232 line receiver)
- 74LS04 (Hex inverter)
- 74LS373 (octal latch)

With these chips, a complete microcomputer system can be built with the following features:

- 2K byte Basic/Debug interpreter in the internal ROM.
- 4K bytes of user RAM.
- 2K bytes of user-programmable EPROM.
- Full-duplex serial operation with programmable baud rates.
- RS232 interface.
- 8-bit counter/timer with associated 6-bit prescalers.
- 124 general-purpose registers internal to the Z8671.
- 14 I/O lines available to the user.
- 3 lines for external interrupts.
- 3 sources of internal interrupts.
- Sophisticated, vectored interrupt structure with programmable priority levels. Each can be individually enabled or disabled, and all interrupts can be globally enabled or disabled.
- External memory expansion up to 124K bytes.
- Memory-mapped I/O capabilities.

This microcomputer can be used as a microcontroller, in which case a terminal is attached, via the RS232 interface, and Basic/Debug is used to create, test, and debug the system. When the system is debugged, the program is put into the EPROM, the terminal disconnected, and the board run standing alone. The terminal can be reat-tached at any time to monitor the subroutines running on the board.

This proposed board meets the design requirements of simplicity and of allowing the user to write and debug programs in Basic while maintaining access to the Z8671 on-chip features.

Interfacing the Z8671 with External Memory

Both RAM and ROM are used in this application for program development and to demonstrate the use of components with and without address latches.

The RAM interface is easy to implement when using a Z6132 (Figure 1). No external address latch is needed because the Z6132 latches the address internally. The Z6132 signals WE (Write Enable), DS (Data Strobe), and AC (Address Clock) are wired directly to the Z8671 signals R/W (Read/Write), DS (Data Strobe), and AS (Address Strobe). The only other signal required is CS (Chip Select). CS is provided by the Z8671 by decoding the upper address bit of port 0. This board uses address bit 15 to select the chip. Since there are two memory chips on this board, the upper address bit ensures that the Z6132 is selected for addresses 800-7FFF (Hex) and that the 2716 is selected by addresses 8000-FFFF (Hex).

There are two major advantages to using the Z6132. The interface to the Z8671 is uncomplicated because both components are Z-BUS™ compatible, and it provides 4K bytes of RAM in one package.

The ROM interface is not as simple as the interface to the Z6132. Nevertheless, the circuit is common in microcomputer applications. The ROM does not latch the address from the Z8671 and therefore needs an external address latch. The 74LS373 latches the address for the 2716 EPROM. The Enable pin on the 74LS373 is driven by the AS signal via an inverter. The EPROM is also selected by the upper address nibble of port 0. Figure 2 shows the Z8671-to-2716 interface.

Interfacing the Z8671 with RS232 Port

The Z8671 uses its serial communication port to communicate with the RS232 port. Driver and receiver circuits are required to supply the proper signals to the RS232 interface. The circuit of Figure 3 shows the interface between the Z8671 and the 1488 and 1489 for serial communication via the RS232 interface.

The serial interface does not use the control signals Clear to Send, Data Set Ready, etc. It uses only Serial In, Serial Out and Ground, so it is a very simple interface.

The Z8671 uses one timer and its associated prescaler for baud rate control. When the Z8671 is reset, it reads location FFFD and uses the byte
Figure 1. The Z8671 and Z6132 Interface

Figure 2. The Z8671 and 2716 Interface
stored there to select the baud rate. The board described in this application note uses EPROM to select the baud rate. On reset, the Z8671 reads FFFD, which is in the EPROM, and decodes the baud rate from the contents of that location. The baud rate can be changed in software.

Figure 4 shows the full board design implemented for this application note.

Uncommitted I/O Pins and Other Pins

Using the above design, port 2 is available for user applications. Any of the port 2 pins can be individually configured for input or output. There are also six pins in port 3 available to the user. The port 3 input pins can be used for interrupts.

SOFTWARE

Getting Started

The Z8671 board needs +5 V and ground to run all components on the board except the 1488 EIA line driver. The 1488 needs -12 V and -12 V in addition to the +5 V and ground. (If using no terminal, the EIA driver/receiver circuit is disconnected. Consequently, the +12 V and -12 V lines are not required.) The test board ran at 200 mA.

The RS232 port can interface to any ASCII terminal if the baud rate setting is matched to the value programmed into the EPROM. With power supplied to the board and the terminal connected to it, the reset button resets the Z8671 and the prompt character appears ("\":").

The board is ready for a Basic command when the ":" appears. The following sequence is a simple I/O example:

```
CS 20
```

Figure 4. The Z8 System with Basic/Debug
When a number is entered as the first character of a line, the Basic monitor stores the line as part of a program. In this example, "10 input a" is entered. Basic stores this instruction in memory and prints another ":" prompt. The Run command causes execution of the stored program. In this example, Basic asked for input by printing ">". A number (5) is typed at the terminal. Basic accepts the number, stores it in the variable "a", and executes the next instruction. The next instruction (20 "a=";a) is an implied print statement; writing an actual "print" command is not necessary here. This line of code produced the output "a=5". The command "list" caused Basic to display the program stored in memory on the terminal.

Reading Directly from Memory

Basic lets the user directly read any byte or word in memory using the Print and "@" for byte references or "@" for word references:

:print @8
 10
:printhex(@8)
 A
:printhex(A@8)
 AF6

The first statement prints the decimal value of Register 8. The next statement prints the hexadecimal value of Register 8 and the last statement prints the hexadecimal value of Register 8 (0AH) and Register 9 (F6H).

Writing Directly to Memory

Basic lets the user write directly to any register or RAM location in memory using the Let command and either "@" or "A".

:@a=%ff
 :4096=255
 :print@10
  255
 :printhex(A%1000)
 FF

The Let command is implied to save memory space but can be included. The first statement loads the hexadecimal value FF into register 10 decimal (1000H). The print commands write to the terminal the values that were put in with the first two instructions.

Memory Environment

Table 1 gives the memory configuration for the Z8671 application example. Chip Select is controlled by the MSB (most significant bit or A15) of port 0. Therefore, the RAM is selected for all addresses between 800H (2048 decimal) and 7FFFH (32767 decimal). Addresses BFF, 18FF, 28FF, 38FF, and 78FF address the same location in RAM in this application because of Modulo 4K. EPROM is selected for all addresses from 8000H to FFFFH and, like the RAM, several addresses point to the same location in the PROM.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-2047</td>
<td>(0-7FF)</td>
<td>Internal ROM</td>
</tr>
<tr>
<td>2048-32767</td>
<td>(800-7FFF)</td>
<td>RAM (Z6132)</td>
</tr>
<tr>
<td>32768-65536</td>
<td>(8000-FFFF)</td>
<td>EPROM (2716)</td>
</tr>
</tbody>
</table>

Switching from RAM to EPROM

Register 8 and Register 9 contain the address of the first byte of a user program or, if there is no program, the address where the Z8671 will put the first byte of a user program. In this application example, when the Z8671 is reset, Register 8 and Register 9 contain 800H, which points into RAM. EPROM is selected by changing the contents of register 8 from 08H to 80H (See Table 2).

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Hex</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>22-23</td>
<td>(16-17)</td>
<td>Current Line Number</td>
</tr>
<tr>
<td>8-9</td>
<td>(8-9)</td>
<td>Address of the First Byte of User Program</td>
</tr>
</tbody>
</table>

For more details on the register assignments, refer to the Pointer Registers—RAM System section of the Z8 Basic/Debug Software Reference Manual.

After the instruction "A8=%8000" is executed, the Z8671 accesses the EPROM on the Basic/Debug Board. The example below shows how to switch from RAM to EPROM. The example uses two separate programs, one in RAM and one in EPROM. The RAM program is listed first, then the EPROM.
Baud Control

The baud rate is selected automatically by reading location FFFDH and decoding the contents of that location when the Z8671 is reset (the Z8 Basic/Debug Software Reference Manual contains the baud rate switch settings in Appendix B). This application example holds the baud rate settings in its EPROM. The least significant bits of location FFFDhex will provide baud rates as follows:

<table>
<thead>
<tr>
<th>Baud Rate Value</th>
<th>Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>150</td>
<td>000</td>
</tr>
<tr>
<td>300</td>
<td>111</td>
</tr>
<tr>
<td>1200</td>
<td>101</td>
</tr>
<tr>
<td>2400</td>
<td>100</td>
</tr>
<tr>
<td>4800</td>
<td>011</td>
</tr>
<tr>
<td>9600</td>
<td>010</td>
</tr>
<tr>
<td>19200</td>
<td>001</td>
</tr>
</tbody>
</table>

After a reset, the baud rate is programmed by loading a new value into counter/timer 0 (see the Z8 Technical Manual, section 1.5.7). A Reset always changes the baud rate back to the rate selected from the contents of location FFFD.

Burning an EPROM

The EPROM contains the baud rate selection byte in location 7FDH. The other locations in memory are used for program storage. See section 6.3 of the Basic/Debug Manual for the format used to store programs in memory. This format is used to store programs in EPROM.

Example

The following is a printout of the game Mastermind written in Basic/Debug.

```
10 @243=7
20 @242=10
30 @241=14
40 x=usr(84):a=@242-1:x=usr(84):b=@242-1
50 x=usr(84):c=@242-1:x=usr(84):d=@242-1
55 """:i=0
100 "guess ":in e,f,g,h
110 i=i+1
300 j=%7f22:k=%7f2a
```

Lines 10 through 50 comprise the random number generator for the program. The three lines:

```
10 @243=7
20 @242=10
30 @241=14
```

initialize counter/timer 1 to operate in modulo-10 count. Refer to the Z8 Technical Manual for complete information on initializing timers.

The "usr(84)" function waits for keyboard input, the ASCII value of the key is returned in a variable with the following command:

```
:10 x=usr(84):"" ""
:15 printhex(x)
:run 5
35
```

In the above example, the program waits at line 10 until keyboard input, in this case the number 5. The input value is stored in ASCII format in the variable "x". The line:

```
40 x=usr(84):a=@242-1:x=usr(84):b=@242-1
```

waits for input, reads the current value of timer 1, subtracts 1 (to get a number between 0 and 9), and stores the number in variable a. Then it waits for keyboard input at the second user function call, reads the current value of timer 1, subtracts 1, and stores the number in variable b. Line 50 of the example program gets two more random numbers and stores them in variables c and d. The four-digit random number is located in variables a, b, c, and d.

Line 300 assigns the location of variable a to variable j and the location of variable e (the
first variable in the guess string) to the variable k. The strategy is to access these variables indirectly and to increment pointers j and k to access the variables.

A colon is used to separate commands on the same line. This is useful in packing the program into a small amount of memory space. The code, however, is harder to read. See section 5 of the Basic/Debug manual for more information on memory packing techniques.

Below is a sample run of the Mastermind program:

:run
(<RETURN> on the keyboard is entered four times here)
guess? 0, 1, 2, 3
right 2 place 0
guess? 4, 5, 6, 7
right 2 place 1
guess? 0, 2, 4, 6
right 3 place 2
guess? 4, 2, 1, 6
right 4 place 4
right in 4 guesses
play another? y/n
?n

CONCLUSION

The design of this application example met the major design goals of simplicity and functionality. The first goal is accomplished by prudent selection of support components, excluding any unnecessary chips. The board allows the user to exercise the full power and flexibility of the features of the Z8601 not used by Basic/Debug. The user can write and debug Basic programs without detailed knowledge of the Z8601.

The Basic application example demonstrates a memory interface that is applicable for all Z8 Family members. The case where there is no address latch on the memory chip was discussed, and an example of how to interface the multiplexed address/data bus of the Z8 Family through an address latch was shown.

The software section explains the memory environment and gives several examples of Basic/Debug. These examples are a good introduction to the board and to Basic/Debug.

The Z8671 is a customized extension of the Z8601 single-chip microcomputer. The simplicity of the Basic application example demonstrates the flexibility of the Z8601 microcomputer in an expanded memory environment.
INTRODUCTION

The Zilog Z8590 Universal Peripheral Controller (UPC) opens up a wide variety of applications for distributed processing. One of the most useful functions of the UPC is to off-load routine processing tasks, such as I/O processing, from the CPU. The advantages of such a distributed processing approach include greater system throughput, more efficient use of system resources, and protocol converters that make different peripherals look the same to the system software. The last advantage is particularly useful where different hardware configurations may be used with the same software. So long as the UPC handles the CPU interface in the same way, the peripheral devices attached to the UPC are transparent to the CPU.

This paper describes a CRT display and keyboard interface circuit that was designed and built by the Zilog Applications Group using the Z8590 UPC in a Z80 system environment. The CRT display function was chosen due to the widespread use of CRT displays in the data processing environment. For further information on the Z8590 UPC refer to the Zilog Data Book, publication number 00-2034-01.

FUNCTIONAL DESCRIPTION

This paper describes the Input/Output (I/O) part of a computer system in its most rudimentary form. Distributed processing is the theme used in this design so that as much of the low-level processing for I/O as possible is performed by the UPC. Figure 1 shows a block diagram of the UPC I/O system.

Figure 1. Block Diagram of the UPC
Single Board Terminal
The display interfaces to a standard video monitor by way of a composite video signal. Characters are represented by dots on a raster scan display in the form of a 5 x 7 matrix. The CPU interface to the UPC can transfer characters on a single byte basis or by a block move. So far as the CPU is concerned, the UPC looks like a serial port when used in single byte mode. This permits the system software to remain virtually the same for a serially-linked terminal or for the UPC. The UPC also provides for programmable cursor control, like that available on a standard terminal, with the control characters being optionally selected by the system software. When the UPC is initialized by the CPU, a bit in the control word can be set to indicate that cursor control characters will follow. The keyboard input is from an ASCII-encoded keyboard that has a strobe to signal a valid character present.

The standard 7-bit ASCII code is supported with the negative-going strobe pulse indicating valid data. The keyboard input is TTL compatible and is not buffered into the UPC.

SYSTEM DESIGN

The UPC I/O project is designed to fit within an existing Z80-based test bed. Therefore, the interface requirements include a Z80-type interface with interrupt capability. Other specifications include:

- Display format of 16 lines by 64 characters
- 5 x 7 dot matrix characters
- Composite video output
- ASCII character input from CPU
- Programmable cursor control
- ASCII keyboard input
- Single +5V operation
- Character or block transfer mode
- Programmable CPU interrupts
- Programmable enable for CRT and keyboard

HARDWARE DESIGN

The hardware design encompasses three basic elements: the Z8590 UPC and processor interface section, the CRT display section, and the keyboard input section.

The Z8590 UPC is treated as a peripheral by the master CPU, in this case a Z80A CPU, and is accessed using the standard Z80 I/O instructions via two ports. One of the two ports is selected depending on the state of the A/D line. If A/D is Low the address pointer is being written to. If A/D is High the register currently addressed by the address pointer is being accessed.

The Z8590 UPC coordinates operation of the display section and the keyboard input with the Z80 CPU. Six bits from Port 1 are used to transfer data from the UPC to the CRT refresh memory. The other two bits are used with bit 7 of Port 2 to form the three bit command word for the CRT controller. Seven bits of Port 2 are used to input ASCII data from the keyboard. Since four of the bits on Port 3 are used for interrupt control, the other four are used for I/O control. Bit 3 of Port 3 is used for the keyboard input strobe. This input generates an interrupt within the UPC when the strobe input goes Low, indicating valid data at the keyboard inputs. Bit 4 of Port 3 is used to control the RAM write pulse coming from the CRT Controller (CRTC) and going to the RAM. When this bit is Low, RAM writes are inhibited for operations such as cursor home and cursor return. Bit 6 of Port 3 is used to generate the Data Strobe (DS) for the CRTC. When DS goes from Low to High, the three command bits are latched into the CRTC. Figure 2 shows the UPC and interface circuitry used.

The heart of the display circuit is the Standard Microsystems CRT-96364B CRT chip. The basic design was derived from the CRT-96364B data sheet by Standard Microsystems Corp. The CRTC contains all the circuitry necessary to generate the video timing pulses and memory address and control signals for the display RAM. The display format is 64 characters per line by 16 lines. This requires a 1024 character memory which is supplied by the 2102 RAM devices. Since 64 ASCII characters are displayed, only six bits of memory are required to store character information. The memory address and write signals are generated by the CRTC under control of the UPC. Data is entered into the display memory by writing a command to the CRTC along with the data. Figure 3 shows the logic used with the CRTC.

Within an 8 x 8 dot character cell provided by the CRT timing, only a 5 x 7 dot character is used. The characters are formed using a 2716 EPROM character generator. The lowest three bits of the 2716 EPROM address inputs from the character row count and come from the CRTC. The next six bits form the character address. Each character is stored in EPROM as eight contiguous bytes. The row count addresses a row (equivalent to a byte) within the character block. Therefore, the character addresses are modulo 8 and take a total of 512 bytes. The CRV output of the CRTC is used to select the cursor pattern in EPROM. When CRV is Low characters are normally displayed. When CRV
Figure 2. 2-UPC CRT Controller, Section I
Figure 3. 2-Bit CRT Controller, Section II
Figure 5. CRT 96364B Timing Waveforms
is High the character is replaced by an underscore.

Five bits of the EPROM output are fed into the 74LS165 shift register. This shift register converts the five column dots into a bit stream for the video output signal. Composite video is generated by merging the video dot stream with the Composite Sync (CSYN) output of the CRTC through a resistor summing network.

The remaining circuitry supplies clocks to various parts of the circuit. Three elements of the 74504 form an oscillator. The output of the oscillator goes to three places. It is divided by twelve by the 74LS92 to form the 1.018 MHz clock required by the CRT-96364B. It is also divided by four by the 74LS73 to provide the 3.054 MHz clock for the UPC. The oscillator output is also ANDed with the Dot Clock Enable (DCE) output of the CRTS and fed into the 74LS161 to form the Dot Character Clock (DCC) pulses. Since a character cell time is eight clock pulses long, the DCC is derived from a divide-by-eight counter. The divide-by-eight counter also loads the shift register at each character time. Figures 4 and 5 show the circuitry and waveforms for the timing and video output circuitry.

The UPC emulates CRT terminal operations by providing keyboard data input to the master CPU as well as CRT output. The keyboard inputs are 7-bit ASCII encoded with TTL level signals. The Strobe Input (STB) is active Low to indicate a valid character at the keyboard data inputs. When STB goes low, an interrupt is generated within the UPC and the data inputs are read.

With this hardware a complete CRT terminal can be constructed at minimal cost to the user with no sacrifice in performance.

SOFTWARE DESIGN

The software design encompasses two areas: the UPC programming and the master CPU interface. The former includes the UPC internal register organization and program initialization. The latter includes the data transfer protocol used between the UPC and the master CPU.

UPC Programming

The specifics of this CRT project will now be discussed, as it is assumed that the reader is familiar with the UPC in general. Of the 256 accessible registers within the UPC, 22 (addresses %F0 through %FF and %00 through %05) are special-purpose control registers defined by the hardware. The remaining 214 registers are general-purpose in nature and are allocated as shown in Figure 6.

![Figure 6. UPC Internal Register Allocation](image)

The Program (PGM) registers (registers %06 through %0F) are general-purpose data manipulation registers. These are the working-set registers used to hold data temporarily and to perform various comparison and calculation functions within the program.

The CPU access registers (%10 through %1F) are used to facilitate communication between the UPC and master CPU. Two bits in the status register, CRT Busy (CRTBSY) and CPU Data Available (CPDAV), are actually semaphores that form the key mechanisms for data interchange. The CRTBSY bit can be set only by the master CPU and can be cleared only by the UPC. The CPDAV bit can be cleared only by the master CPU and can be set only by the UPC. These will be discussed in detail in the master CPU access section.

A line of data on the CRT screen is 64 bytes long. Therefore registers %20 through %5F form a 64 byte line buffer for the CRT display. This is used only in Block Transfer mode, since the UPC receives a block of data before outputting it to the CRT.

The parameter area (registers %60 through %7F) contains the cursor control characters and corre-
sponding information. Figure 7 illustrates the format of the parameter area. Since there are eight cursor control characters and each occupies four bytes of control block information, there are a total of 32 bytes allocated for this purpose. Most incoming control characters are compared with the ASCII codes in this table, and if a match is found the software determines what to do based on the other values in the cursor control block.

**Figure 7. UPC Parameter Block Definition**

The keyboard buffer (registers %80 through %BF) temporarily stores data coming from the keyboard within the UPC until the master CPU reads the data. The keyboard buffer is used in both character and block modes since keyboard input is actually done by interrupts. In character mode, the buffer is simply a circular buffer that accumulates keyboard data until it is processed by the master CPU. One pointer, the Keyboard Buffer Pointer (KBBPTR), is used to indicate into which location the next keyboard character will go. The other pointer, the Keyboard Pointer (KBPTR), is used to indicate which location the next character will be read from by the master CPU.

Finally, the stack and data areas (registers %C0 through %EF) are used for variable storage. The stack grows down from location %F0 and occupies about ten bytes maximum. The internal data area contains various run-time variables used by the UPC program, as shown in Table 1.

On power-up the UPC initializes the necessary variables, all the control registers, and loads the default parameters into the parameter area. When all this is done the UPC sets the Enable Data Transfer (EDX) bit in the Data Transfer Control (DTC) register. This enables communication with the master CPU to take place, and indicates to the master CPU that the UPC is ready for operation. If the EDX bit is cleared, data transfers to or from the UPC are inhibited. At this point the UPC waits for the Mode register to be set by the master CPU before continuing.

<table>
<thead>
<tr>
<th>UPC ADDRESS</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>%C0</td>
<td>FLAG</td>
</tr>
<tr>
<td>%C1</td>
<td>UBPTR</td>
</tr>
<tr>
<td>%C2</td>
<td>CBCNT</td>
</tr>
<tr>
<td>%C3</td>
<td>COLCNT</td>
</tr>
<tr>
<td>%C4</td>
<td>TIMER</td>
</tr>
<tr>
<td>%C5</td>
<td>KBPTR</td>
</tr>
<tr>
<td>%C6</td>
<td>KBBPTR</td>
</tr>
<tr>
<td>%C7</td>
<td>CHAR</td>
</tr>
</tbody>
</table>

**Table 1. Internal Data Area**

Appendix A contains the UPC program listing used for this project. The UPC program structure consists of constants declaration, the main program body, and data tables. Within the main program body are routines for initialization, the main program loop, CRT output, keyboard input, interrupt service, and other support routines.

**Master CPU Interface**

The master CPU communicates with the UPC through 20 special registers. These registers are accessed directly by the I/O instruction address in the Z8090 Z-UPC and indirectly by a register pointer in the Z8590 UPC. To read or write data for a particular register in the Z8590, the register pointer is first written (A/D line is Low) and then data (A/D line is High) is written. Thus, a register access operation involves two I/O transactions. The register pointer is latched within the UPC so multiple reads of a particular register (such as the status register) need not have the pointer written each time. This is useful when polling the status bits or using a block move instruction for data transfers.

Of the twenty possible registers accessible to the master CPU only ten are actually used. Figure 8 shows eight of these registers and their meanings. The Mode register (register pointer address %00), end-of-line edit character (EOL, %04), backspace edit character (BS, %05), delete-line edit character (DL, %06), and interrupt vector (VECT,
are initialized once by the master CPU. The status, CRT data (CRDAT), and keyboard data (KBDAT) registers are used to control data flow into and out of the UPC.

---

The master interrupt control register (MIC) is used by the master CPU to control the UPC interrupt condition. The upper three bits (D7, D6, and D5) correspond to Interrupt Enable (IE), Interrupt Under Service (IUS), and Interrupt Pending (IP), respectively, by a master CPU read. When the CPU writes these bits, their meanings change as illustrated in the table of Figure 9. The EDX bit (bit 3) is monitored by the CPU after power-up so the CPU can determine when to initialize the UPC.

The interrupt vector register is accessed by writing a 07 hex to the UPC address port, and the vector to the UPC data port.

Next comes the mode control byte. The lower four bits determine the operation of the UPC environment. If CRT Enable (bit 0) is set, then data transfers can occur from the master CPU to the CRT display. If KB Enable (bit 1) is set, then data transfers are enabled from the keyboard to the master CPU. The block mode bit (bit 2) indicates block transfer mode. This applies to both the CRT output and keyboard input. Block mode is used with the powerful ZBO block I/O instructions or with DMA.

The Parameters Follow bit (bit 3) indicates whether or not eight cursor control parameter bytes will follow. If the Parameters Follow bit is set, then the next eight bytes sent to the UPC are the eight cursor control characters in the following sequence: cursor home, cursor forward, cursor back, cursor down, erase page, cursor return, cursor up, and erase line. These eight bytes are written via the DIND register. The DIND register eight cursor control bytes are sent to the UPC data port by a block move instruction (OTIR) on the ZBO.

This completes initialization of the UPC by the master CPU. Listings found in Appendix B can be used as an example of how the master CPU uses the UPC.

---

Using the UPC

Of the ten registers utilized by the master CPU, four or five are actually used for data transfer. The status register (address 01 hex) contains two bits that indicate the internal UPC status. These
bits are monitored and controlled by the master CPU under the definition of the UPC interface protocol. The CRTBSY (bit 0) can be set only by the master CPU and cleared only by the UPC. When the master CPU writes data into the CRT Data register (CRDAT, address 02 hex), it also sets the CRTBSY bit in the status register. This does two things. First, it indicates to the UPC that there is data available in the CRDAT register ready to output to the CRT display. Second, the busy bit remains set and prevents further character transfers until the UPC clears the busy bit. Figure 10 shows the data flow for character mode transfers into and out of the UPC.

Similar to the CRT data transfer is the keyboard data transfer. The keyboard data register (KBDAT, address 03 hex) contains the keyboard data loaded by the UPC, and the CPDAV bit in the status register (bit 1) indicates keyboard data is available. The CPDAV bit can be set only by the master CPU. When the master CPU reads KBDAT, it also clears CPDAV in the status register. This is also shown in Figure 10. The sequence of events depicted in Figure 10 is important. The order in which the registers are accessed should be adhered to or the UPC may change or lose data unexpectedly.

The above description applies to character transfers when polling the status register continuously. Interrupts can be used with the UPC to indicate a change in either status bit. If CPDAV goes from a 0 to a 1 (set) or CRT busy goes from a 1 to a 0 (cleared) the UPC generates an interrupt. The interrupt service routine must poll the status register to determine the cause of the interrupt, however, since there is only one vector returned in vectored interrupt mode.

If interrupts are used, then the master CPU interrupt service routine must perform several operations in addition to the data transfer(s). These operations involve the Master Interrupt Control (MIC) register (address 1E hex). After the data transfer condition has been satisfied in the UPC the master CPU must reset the IP and IUS latches within the UPC. This restores the daisy chain to its normal state. Then, to allow further interrupts from the UPC, the IE latch must be set. Using bits D7, D6, and D5 of the MIC register (shown in Figure 9), IP and IUS are cleared by writing 001. IE is then set by writing 110 to these bits. IE is cleared by the UPC on power-up, thus the set IE command must be written to the UPC during the initialization phase by the master CPU so that interrupts can occur. The interrupt operation applies to both character mode transfers and block mode transfers.

Block mode data transfers are faster and more efficient than character mode transfers. These transfers access the status register, as do character transfers, but the data is exchanged via the DIND register. DIND is a location pointed to by another register within the UPC. Master CPU accesses to DIND automatically increment the pointer register by one so that several consecutive register locations can be written to or read from. The number of bytes to transfer by DIND is written by the master CPU into CRDAT for CRT block transfers, and read from KBDAT for keyboard block transfers. Thus, protocol exists for CRT block data transfers, as Figure 11 illustrates. Up to 64 bytes may be sent or received at one time in this mode. Both the Z80 and Z8000 block move instructions work very well with this method of data transfer, resulting in superior system throughput.

Using the Z8090 Z-UPC

Implementing the single board terminal on a Z8000 or Z8 processor-based system is very easy with the Z8090 Z-UPC. The software in the Z-UPC is identical to the software in the Z8590 UPC. The hardware interface to the keyboard and display cir-
Block Mode (transfer handshake)

<table>
<thead>
<tr>
<th>CPU</th>
<th>UPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read CRTBSY</td>
<td>CRTBSY = x</td>
</tr>
<tr>
<td>If set, Loop</td>
<td>CRTBSY = 0 (IP set if CRTBSY was 1)</td>
</tr>
<tr>
<td>Write block length</td>
<td>CRTBSY = 1</td>
</tr>
<tr>
<td>Set CRTBSY</td>
<td>CRTBSY = 0, set IP</td>
</tr>
<tr>
<td>Loop if set</td>
<td>CRTBSY = 1</td>
</tr>
<tr>
<td>Block output data</td>
<td>Process data</td>
</tr>
<tr>
<td>Set CRTBSY</td>
<td>CRTBSY = 1</td>
</tr>
</tbody>
</table>

Figure 11. Block Mode Data Output to UPC

...circuitry is also the same. The only difference is the hardware interface to the CPU and the CPU software. The protocol and register functions are unchanged.

CONCLUSION

This paper describes the use of the Z8590 UPC in a distributed processing environment. System performance can be most effectively improved by dividing CPU tasks into logical functions. Such a task, as has been illustrated here, is a fundamental I/O operation that facilitates communication between the user and the computer. Other functions may include such peripheral operations as a flexible disk controller, a PROM programmer, a D/A or A/D converter, or a communications protocol controller.

Coupled with the powerful instruction set of the Zilog family CPUs, the Z8090 Z-UPC and Z8590 UPC find many uses in virtually any system environment.
UPC CRT Controller Program Listing

! UPC CRT TERMINAL DRIVER PROGRAM!

CRTC MODULE

CONSTANT

DTC:=0 !DATA XFER CONTROL REG!
P1:=1 !PORT 1!
P2:=2 !PORT 2!
P3:=3 !PORT 3!
LC:=4 !LIMIT COUNT REG!
DIND:=5 !DATA INDIRECTION REG!
TMVAL:=%28 !TIMER COUNT VALUE!
DSC:=%10 !CPU ACCESS AREA!
MODE:=DSC !MODE REGISTER!
CRTEN:=1 !CRT ENABLE BIT!
KBEN:=2 !KB ENABLE BIT!
BLOK:=4 !BLOCK XFER!
PARMS:=8 !PARAMETERS FOLLOW!
STAT:=MODE+1 !STATUS REGISTER!

CRBSY:=1 !CRT BUSY FLAG!
CPDAV:=2 !CPU KB DATA AVAIL!
KBOVF:=4 !KB BUFFER OVERFLOW!
CRDAT:=STAT+1 !CRT DATA AREA!
KBDAT:=CRDAT+1 !KB DATA AREA!
EDL:=KBDAT+1 !END OF LINE CHARACTER!
BS:=EDL+1 !BACKSPACE CHARACTER!
DL:=BS+1 !DELETE LINE CHARACTER!
VECT:=DL+1 !CPU INTERRUPT VECTOR!
BUFF:=%20 !CRT BUFFER AREA!
PARAM:=%60 !PARAMETER TABLE AREA!
KBUFF:=%80 !KEYBOARD INPUT BUFFER!
STOR:=%C0 !RAM STORAGE AREA!
FLAG:=STOR !FLAG BYTE!
KBB:=1 !KB BUFFER OVF FLAG!
KBDAV:=2 !KB DATA AVAIL!
CRTXF:=4 !CRT XFER FLAG!
KBXFR:=8 !KB XFER FLAG!
TMRLQ:=%B0 !TIMER ACTIVE FLAG!
UBPTR:=FLAG+1 !UPC CRT BUFFER POINTER!
CBCNT:=UBPTR+1 !CPU CRT BYTE COUNT!
COLCNT:=CBCNT+1 !CRT COLUMN COUNT!
KBPTR:=COLCNT+1 !KB OUTPUT BUFFER PTR!
KBBPTR:=KBPTR+1 !KB INPUT BUFFER PTR!
TIMER:=KBBPTR+1 !TIMER VALUE!
CHAR:=TIMER+1 !KB CHARACTER STORAGE (KLUGE)!
MIV:=%F0 !CPU INTERRUPT VECTOR REG!
MIC:=%C8 !MASTER INTERRUPT CTRL!
EDX:=8 !ENABLE DATA XFER BIT!
IP:=%C0 !SET IP BIT!
DEL:=%D !DEFAULT EOL!
DBL:=%B !DEFAULT BACKSPACE!
TMR:=%B !DEFAULT DEL LINE!

SECTION PROGRAM

GLOBAL

*ABS 0

P 0000 0290 WVAL ERROR
P 0002 0219 WVAL KBINT
P 0004 0293 WVAL DUMMY
P 0006 0293 WVAL DUMMY
P 0008 0206 WVAL TIMER
P 000A 0218 WVAL TIMER
P 000C

MAIN PROCEDURE

ENTRY

BEGIN:

P 000C BF DI
P 000D BO FD CLR RP CLEAR REGISTER POINTER
P 000F B0 C0 69 CLR FLAG CLEAR FLAG BYTE!
P 0011 B0 C7 70 CLR CHAR CLEAR CHARACTER!
P 0013 B0 C6 71 CLR TIMER CLEAR TIMER!
P 0015 B0 10 72 CLR MODE CLEAR MODE!
P 0017 B0 11 73 CLR STAT CLEAR STATUS!
P 0019 E6 C8 B0 74 LD KBBPTR.#KBUFF INIT KBBPTR!
P 001C E6 C4 B0 75 LD KBBPTR.#KBUFF!
P 001F E6 14 0D 76 LD EOL.#EOL !DEFAULT EOL=CR!
P 0022 E6 15 08 77 LD BS.#BS !DEFAULT BS=BS!
P 0025 E6 16 18 78 LD DL.#DL !DEFAULT DL=LINE=CAN!
P 0028 E6 00 10 79 LD DTC.#DSC LOAD DTC REG.!
P 002B 6C 60 80 81 LD R6.!L7.FO !POINT TO REGS.!
P 002F 8C 02 82 LD R8.!HI CCTABL SOURCE!
P 0031 9C A4 83 LD R9.!LO CCTABL!
P 0033 C3 68 85 LOCI C!R6.C!RR8 MOVE BYTES'
P 0035 7A FC 86 D.JNZ R7.CLOOP!
P 0037 8C 02 87 LD R8.!HI TABLE!LOAD 8 BYTES!
P 0039 9C 94 88 LD R9.!LO TABLE!
P 003B 6C FO 89 LD R6.!L7.FC !POINT TO REGS.
P 003D 7C 10 90 LD R7.!HI TABLE!
P 003F C3 68 92 LOCI C!R6.C!RR8 MOVE INIT CODES.
P 0041 7A FC 93 D.JNZ R7.ILOOP TO REGS.!
P 0043 44 10 10 95 OR MODE.MODE !MODE WORD SET?!
P 0046 68 FB 96 JR Z.ML !NO. LOOP!
P 0048 E4 17 F0 97 LD MIV.VECT !SAVE CPU INTR VECTOR!
P 004B 76 10 08 98 TM MODE.#PARMS !CHECK PARMS BIT!
P 004E 68 18 99 JR Z.SKIP !SKIP IF CLEAR!
P 0050 E6 05 20 100 LD DIND.#BUFF!
P 0053 E6 04 0B 101 LD R7.!LO CCTABL!
P 0056 44 04 04 103 OR LC.LC !WAIT FOR LC=0!
P 0059 E8 F8 104 JR NZ.ML!
P 005B 6C 08 105 LD R6.#B !MOVE 8 BYTES!
P 005D 7C 60 106 LD R7.#PARAM!
P 005F BC 20 107 LD R8.#BUFF!
P 0061 E3 9B 109 LD R9.#R9!
P 0063 F3 79 110 LD @R7.R9!
P 0065 06 E7 04 111 ADD R7.#4!
P 0068 BE 112 INC R8!
P 0069 6A F6 113 DJNZ R6.ML2!
P 006B 9F 115 EI!
P 006C 76 10 01 117 ! THIS IS THE MAIN PROGRAM LOOP.
P 0070 76 11 01 118 UPC ARRIVES HERE AFTER INIT AND
P 0074 6B 03 119 MODE ARE DEFINED.
P 0076 D6 0094 120 !
P 0078 C6 10 01 122 LOOP:
P 007A B6 0B 123 TM MODE.#CRTEN !CRT ENABLED?!
P 007C 10 02 124 JR Z.L1 !NO. BRANCH!
P 007E 76 C5 00 125 TM STAT.#CRTBSY !CRT DATA AVAILABLE?!
P 0080 6B 03 126 JR Z.L1!
P 0082 D6 0094 127 CALL CRT!
P 0084 76 10 02 128 L1:
P 0086 76 11 01 129 TM MODE.#KBEN!
P 0088 EE 130 JR Z.LOOP!
P 008A 76 C0 02 131 TM FLAG.#KBDAV KB DATA AVAILABLE?!
P 008C 6B 03 132 JR Z.L2 !NO. BRANCH!
P 008E D6 00DB 133 CALL KB !CHECK KB DATA!
P 0090 B0 C7 134 L2:
P 0092 DB 135 OR CHAR.CHAR !ECHO CHAR?!
P 0094 E1 136 JR Z.LOOP !NO. BRANCH!
P 0096 6B C7 137 LD R6.CHAR!
P 0098 D6 014C 138 CALL DATOUT!
P 009A B0 C7 139 CLR CHAR!
P 009C 8B DB 140 JR LOOP!
P 009E C7 C7 142 ! THIS ROUTINE PROCESSES CRT CHARACTERS THAT
P 00A0 C7 C7 143 ARRIVE FROM THE MASTER CPU.

1-98
145 INPUTS: NONE
146 R6=R10 USED
147 OUTPUTS: NONE
148 !
149 CRT:

P 0094 76 10 04 150 TM MODE,#BLOK !CHECK MODE!
P 0097 6B 37 151 JR Z.CRT3 !BRANCH IF NOT BLOCK!
P 0099 76 C0 08 152 TM FLAG,#KBXF R !KB XFER? !
P 009C 76 06 153 JR NZ,CRT4 !YES, BRANCH!
P 009E 76 C0 04 154 TM FLAG,#CRTXF R !CHECK XFER FLAG!
P 00A1 EB 1C 155 JR NZ,CRT2 !BRANCH IF BLOCK DATA!
P 00A3 6B 12 156 LD R6,CRDAT !GET DATA!
P 00A5 56 E6 3F 157 AND R6,#3F !ONLY 64 BYTES!
P 00AB 6B 0D 158 JR Z.CRT1 !BRANCH IF NOTHING!
P 00AA 69 04 159 LD LC,R6 !MOVE BYTE COUNT!
P 00AC 69 C2 160 LD CBCNT,R6
P 00AE E6 C1 20 161 LD UBPRTR,#BUFF !RESET BUFFER PTR!
P 00B1 E6 05 20 162 LD DIND,#BUFF !SET DIND PTR!
P 00B4 46 C0 04 163 OR FLAG,#CRTXF R !SET XFER FLAG!

164 CRT1:

P 00B7 56 11 FE 165 AND STAT,#%FF-CRTBSY !CLEAR CRT BUSY
P 00BA 46 FE 20 166 OR MIC,#IP !ELSE, SET IP!
P 00BD BB 1B 167 JR CRT4

168 CRT2:

P 00BF E5 C1 E6 169 LD R6,UBPRTR !SET DATA!
P 00C2 D6 014C 170 CALL DATOUT !SEND TO CRT!
P 00C5 20 C1 171 INC UBPRTR
P 00C7 00 C2 172 DEC CBCNT !DEC BYTE COUNT!
P 00C9 E8 F4 173 JR NZ,CRT2 !BRANCH IF MORE!
P 00C8 56 C0 FB 174 AND FLAG,#%FF-CRTXF R !CLR XFER FLAG!
P 00CE BB E7 175 JR CRT1 !EXIT!

176 CRT3:

P 00D0 68 12 177 LD R6,CRDAT !GET DATA!
P 00D2 D6 014C 178 CALL DATOUT !SEND TO CRT!
P 00D5 BB E0 179 JR CRT1 !EXIT!

180 CRT4:

P 00D7 AF 181 RET

182

183 !

184 !

185 !

186

187 KB:

P 00DB 76 10 04 188 TM MODE,#BLOK !BLOCK MODE?!
P 00DB 6B 41 189 JR Z.KB2 !NO, BRANCH!
P 00DD 76 C0 04 190 TM FLAG,#CRTXF R !CRT XFER? !
P 00EO EB 6B 191 JR NZ,KB4 !YES, BRANCH!
P 00E2 76 C0 08 192 TM FLAG,#KBXF R !XFER SET?!
P 00E5 EB 26 193 JR NZ,KB2 !YES, BRANCH!
P 00E7 BF 194 DI

P 00EB 56 11 FB 195 AND STAT,#%FF-KBDVF !CLEAR KB OVF!
P 00EB 76 C0 01 196 TM FLAG,#KB !CHECK KB!
P 00ED E6 06 197 JR Z.KB !CLEAR KB!
P 00F0 46 11 04 198 OR STAT,#KBDVF !SET KB OVF!
P 00F3 56 C0 FE 199 AND FLAG,#%FF-KBB !CLEAR KB!

200 KB1:

P 00F6 68 C5 201 LD R6,KBBPTR !GET LINE LENGTH!
P 00F8 26 E6 80 202 SUB R6,#BUFF
P 00FB 69 13 203 LD KBDAT,R6 !STORE COUNT!
P 00FD 69 04 204 LD LC,R6 !STORE BUFFER LENGTH!
P 00FF E6 05 B0 205 LD DINT,#BUFF
P 0102 46 C0 08 206 OR FLAG,#KBXF R !SET XFER!
P 0105 46 11 02 207 OR STAT,#CPDAV !SET CP DAV!

208 KB11:

P 0108 46 FE 20 209 OR MIC,#IP !SET IP!
P 0108 B8 3D 210 JR KB4

211 KB2:

P 010D 76 11 02 212 TM STAT,#CPDAV !CPU THRU? !
P 0110 EB 3B 213 JR NZ,KB4 !NO, CONTINUE!
P 0112 BF 214 DI

P 0113 56 C0 F7 215 AND FLAG,#%FF-KBXF R !ELSE, CLEAR XFER!
P 0116 E6 C4 80 216 LD KBPTR,#BUFF !RESET KB PTR!
P 0119 E6 C5 80 217 LD KBBPTR,#BUFF
P 011C BB 29 218 JR KB32

219 KB3:

P 011E 76 11 02 220 TM STAT,#CPDAV !CP DAV ?!
THIS ROUTINE OUTPUTS DATA TO THE CRT, IF DISPLAYABLE, ELSE TRANSLATES THE CODE INTO CONTROLLER FUNCTION.

INPUTS: XR6-ASCII DATA XR7-XR10 USED OUTPUTS: NONE

DATOUT:
P 014C A6 E6 20 253 CP R6,%20 !CTRL CHAR ?!
P 014F FB 53 254 JR NC.CHROUT !NO, BRANCH!
P 0151 A6 E6 09 255 CP R6,%9 !TAB ?!
P 0154 6B 41 256 JR Z.DAT2 !YES, BRANCH!
P 0156 AC 60 257 LD R9,%PARAM !POINT TO PARAM TABLE!
P 0158 AC 08 258 LD R10,%8

DATO:
P 015A A3 69 260 CP R6,%R9 !CHECK DATA AGAINST...
P 015C 6B 08 261 JR Z.DAT1 !...CTRL TABLE VALUES!
P 015E 06 E9 04 262 ADD R9,%4

DAT1:
P 0162 00 EA 263 DEC R10

DAT5:
P 0165 AF 264 JR NZ.DAT0 !LOOP UNTIL...
P 0167 9E 265 RET !EXIT IF NO MATCH!
P 0169 9E 266 INC R9

DAT3:
P 016D 76 E7 40 272 TM R7,%40 !INCR COLCNT ?!
P 0170 6B 0E 273 JR Z.DAT11 !ND, BRANCH!
P 0172 20 C3 274 INC COLCNT

DAT4:
P 0174 56 C3 3F 275 AND COLCNT,%3F !EOL ?!
P 0177 EB 1A 276 JR NZ.DAT5 !ND, BRANCH!
P 0179 E3 B9 277 LD R8,%R9 !LOAD SCROLL DELAY VAL!
P 017B 67 E7 08 278 OR R7,%B !SET WRITE ENABLE!
P 017E BB 13 279 JR DAT9 !OUTPUT CTRL CODE!
P 0180 76 E7 10 280 TM R7,%10 !CLEAR COLCNT ?!
P 0183 B3 04 281 JR Z.DAT12 !ND, BRANCH!
P 0185 B0 C3 282 CLR COLCNT

DAT2:
P 0187 BB 0A 283 JR DAT5

DAT6:
P 0189 76 E7 20 286 TM R7,%20 !DECR COLCNT?!
P 018C 6B 05 287 JR Z.DAT5 !ND, BRANCH!
P 018E 00 C3 288 DEC COLCNT

DAT7:
P 0190 56 C3 3F 289 AND COLCNT,%3F !MODULO 64!
P 0193 6C 00 290 LD R6,%0

DAT8:
P 0195 BB 27 291 JR OUTP !OUTPUT TO CRT!
P 0197 6C 20 292 LD R6,%20 !LOAD SPACE!
P 0199 D6 01A4 293 CALL CHROUT !DATA TO CRT!
P 019C 6B C3 296 LD R6, COLCNT !CHECK COLUMN COUNT!
P 019E 56 E6 07 297 AND R6,#7 !MODULO 8?!
P 01A1 EB F4 298 JR NZ,DAT2 !NO. LOOP!
P 01A3 AF 299 RET 300

301 ! THIS ROUTINE OUTPUTS A DISPLAYABLE CHARACTER TO THE CRT. IF COLCNT = EOL (64) THEN DELAYS FOR SCROLL. ELSE, NO DELAY.
304 !
305
306 CHROUT:

P 01A4 B0 EB 307 CLR R8 !INIT DELAY VALUE!
P 01A6 20 C3 308 INC COLCNT
P 01A8 56 C3 3F 309 AND COLCNT, #3F !MODULO 64!
P 01AB BC 04 310 JR NZ,CROUT1

312 CROUT1:

P 01AF 26 E6 20 313 SUB R6,#20 !REMOVE ASCII BIAS!
P 01B2 7C OF 314 LD R7,XOF !CRTC COMMAND!
P 01B4 D6 01 BE 315 CALL OUTP !DATA TO CRT!
P 01B7 BC 07 316 LD R11,X7 !DELAY CHAR TIME!

317 CROUT2:

P 01B9 00 EB 318 DEC R11
P 01BB EB FC 319 JR NZ,CROUT2
P 01BD AF 320 RET 321

322 ! THIS ROUTINE DOES THE ACTUAL DATA WRITE TO THE CRT CONTROLLER CHIP.
323
324
325 INPUTS: %R6=ASCII DATA
326 %R7=CRTC COMMAND
327 %R8=TIMER DELAY VALUE
328 %R9-%R10 USED
329
330 OUTPUTS: NONE
331 !
332
333 OUTP:

P 01BE 76 CO B0 334 TM FLAG,#TMRFLG !CHECK TIMER FLAG!
P 01C1 98 F9 335 JR NZ,OUTP !DROP IF BUSY!
P 01C3 56 C3 EF 336 AND P3,#XEF !CLEAR WRITE ENABLE!
P 01C6 76 E7 0B 337 TM R7,#8 !WRITE ENABLE?!

338 OUT1:

P 01C9 68 03 338 JR Z.OUT1 !NO. BRANCH!
P 01CB 46 03 10 339 OR P3,#X10 !RAM WRITE ENABLE!

340 OUT2:

P 01CE 56 E6 3F 341 AND R6,#X3F !MASK UPPER BITS!
P 01D1 6B E7 342 LD R9,R7 !MASK LOWER 3 BITS!
P 01D3 56 E9 07 343 AND R9,#7
P 01D6 EO E9 344 RR R9
P 01DB EO E9 345 RR R9
P 01DA AE E9 346 LD R10,R9 !MERGE COMMAND BITS!
P 01DC 56 EA CO 347 AND R10,#X0
P 01DF 42 6A 348 OR R6,R10
P 01E1 69 01 349 LD P1,R6 !OUTPUT DATA & CMD!
P 01E3 69 E9 350 RR R9 !GET UPPER CMD BIT!
P 01E5 56 E9 B0 351 AND R9,#X80
P 01EB 56 02 7F 352 AND P2,#X7F !CLEAR COMMAND BIT!
P 01EB 44 E9 02 353 OR P2,R9 !WRITE UPPER CMD BIT!
P 01EE B6 03 40 354 XOR P3,#X40 !GENERATE DS!
P 01F1 B6 03 40 355 XOR P3,#X40
P 01F4 42 88 356 OR R6,R8 !ZERO TIMER VALUE?!
P 01F6 68 0D 357 JR Z.OUT2 !YES, SKIP!
P 01FB 89 C6 358 LD TIMER,R8 !LOAD TIMER!
P 01FA 46 C0 80 359 OR FLAG,#TMRFLG !FLAG TIMER BUSY!
P 01FD D6 E4 28 360 LD TO,#TMRVAL !LOAD TIME CONSTANT!
P 0200 46 F1 03 361 OR THR,#3 !START TO!
P 0203 00 C6 362 DEC TIMER

363 OUT2:

P 0205 AF 364 RET 365

366 ! * INTERRUPT ROUTINES * !
367
368 TIMERO:

P 0206 44 C6 C6 369 OR TIMER,TIMER !SEE IF TIME DONE!
P 0209 6B 09 370 JR Z.DELAY1 !BRANCH IF DONE!
P 020B 66 F4 28 371 LD TO,#TMRVAL !ELSE, RESET TIMER!
P 020E 46 F1 03 372 OR THR,#3 !LOAD & ENABLE TIMER!
P 0211 00 C6 373 DEC TIMER !BUMP TIME COUNT!
P 0213 BF 374 IRET
375 DELAY1:
P 0214 56 CO 7F 376 AND FLAG.#%FF-TMRFLG!CLEAR TIMER BUSY FLAG!
P 0217 BF 377 IRET
378
379 TIMER1:
P 0218 BF 380 IRET
381
382 KB.INT:
P 0219 FB 02 383 LD R15.P2 !GET KB CHAR!
P 021B 56 EF 7F 384 AND R15.#%7F !MASK UPPER BIT!
P 021E 76 C0 01 385 TM FLAG.#KB !KBB SET?!
P 0221 EB 33 386 IRET
P 0222 76 10 04 387 TM MODE.#BLOK !BLOCK MODE?!
P 0225 6B 33 388 JR Z.KB13 !NO. BRANCH!
P 0228 76 11 02 389 TM STAT.#CPDAV !CPDAV?!
P 022B 33 38C .JR NZ.KB14 !NO. BRANCH!
P 022D F9 C7 391 LD CHAR.R15 !ECHO TO CRT!
P 022F A4 14 EF 392 CP R15.EOL !EOL?!
P 0232 FB 10 393 JR NZ.KB15 !NO. BRANCH!!
P 0235 6B 44 395 .JR Z.KB16 !YES, BRANCH!
P 0238 6B 4E 397 .JR Z.KB17 !YES, BRANCH!
P 023B F5 EF C5 398 LD #KBBPTR.R15 !STORE CHAR!
P 0241 20 39A INC KBPPTR !BUMP KBPPTR!
P 0244 66 C5 3F 400 AND KBPPTR.#%3F
P 0246 46 C5 80 401 OR KBPPTR.#KBUFF
P 0249 A4 C4 C5 402 CP KBPPTR.KBPR !EOB?!
P 024C EB 41 403 JR NZ.KB17 !NO. BRANCH!
P 024E 46 C0 02 404 OR FLAG.#KBDAV !SET KB DAV!
P 0250 F5 EF C5 398 LD #KBBPTR.R15 !STORE CHAR!
P 0253 FB 10 409 JR KB17
P 0255 BB 34 410 JR KB17
P 0258 F5 EF C5 412 LD #KBBPTR.R15 !STORE CHAR!
P 025E 20 C5 413 INC KBPPTR
P 0260 56 C5 3F 414 AND KBPPTR.#%3F
P 0263 66 C5 80 415 OR KBPPTR.#KBUFF
P 0266 46 C0 02 416 OR FLAG.#KBDAV !SET KB DAV!
P 0269 A4 C4 C5 417 CP KBPPTR.KBPR !EOB?!
P 026C 6B E3 418 JR Z.KB12 !YES, BRANCH!
P 026E 20 C5 419 JR KB17
P 0270 F5 EF C5 421 LD #KBBPTR.R15 !STORE CHAR!
P 0273 20 C5 422 INC KBPPTR
P 0275 56 C5 3F 423 AND KBPPTR.#%3F
P 0278 46 C5 80 424 OR KBPPTR.#KBUFF
P 027B BB D9 425 JR KB11
P 027D A4 C4 C5 427 CP KBPPTR.KBPR !EOB?!
P 0280 6B 0D 428 JR Z.KB17 !YES, SKIP!
P 0282 00 C5 429 DEC KBPPTR
P 0284 56 C5 3F 430 AND KBPPTR.#%3F
P 0287 46 C5 80 431 OR KBPPTR.#KBUFF
P 028A BB 03 432 JR KB17
P 028C E6 C5 80 434 LD KBPPTR.#KBUFF !RESET KBPPTR!
P 028F BF 436 IRET
P 0290 EB 00 438 ERROR:
P 0292 BF 439 LD R14.DTC !CLEAR ERROR BITS!
P 0294 0000 440 IRET
441 DUMMY:
P 0296 BF 442 IRET
P 0298 BF 443 IRET
444 !REGISTER DATA TABLE FOR INITIALIZATION!
P 029A 0000 445 RVAL %0000
CURSOR CONTROL DEFAULT PARAMETER TABLE

SETUP AS FOLLOWS:

BYTE 1 - ASCII CHAR CODE
BYTE 2 - CRT CODE
BYTE 3 - NOT EOL DELAY VALUE
BYTE 4 - EOL DELAY VALUE (FOR SCROLL)

CURSOR CONTROL DEFAULT PARAMETER TABLE

CCTABL:

CURSOR HOME!
CURSOR FORWARD!
CURSOR BACK!
CURSOR DOWN!
CURSOR RETURN!
CURSOR UP!
PAQE ERASE!
ERASE LINE!

Assembly complete
APPENDIX B

Z80 Test Program Listings for SBT

UPC.INIT

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ CODE</th>
<th>M STATMT</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>;</td>
<td>;</td>
<td>ASM 5.9</td>
</tr>
<tr>
<td>2</td>
<td>;</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>KEQ</td>
<td>-1</td>
<td>KB INPUT ENABLE SW.</td>
</tr>
<tr>
<td>4</td>
<td>CRTEQ</td>
<td>-1</td>
<td>CRT OUTPUT ENABLE SW.</td>
</tr>
<tr>
<td>5</td>
<td>INTEN</td>
<td>0</td>
<td>INTERRUPT ENABLE SW.</td>
</tr>
<tr>
<td>6</td>
<td>BLOCKE</td>
<td>-1</td>
<td>BLOCK MOVE ENABLE SW.</td>
</tr>
<tr>
<td>7</td>
<td>PRME</td>
<td>-1</td>
<td>PARAMETERS TEST SW.</td>
</tr>
<tr>
<td>8</td>
<td>RAM</td>
<td>2000H</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>DPQ</td>
<td>10H</td>
<td>UPC PORT ADDR</td>
</tr>
<tr>
<td>10</td>
<td>DPORT</td>
<td>CPOR+1</td>
<td>UPC DATA PORT</td>
</tr>
<tr>
<td>11</td>
<td>DTC</td>
<td>18H</td>
<td>DTC CONTROL REGISTER</td>
</tr>
<tr>
<td>12</td>
<td>DIND</td>
<td>15H</td>
<td>DATA INDIRECTION REG</td>
</tr>
<tr>
<td>13</td>
<td>MIE</td>
<td>1EH</td>
<td>MASTER INT CONTROL</td>
</tr>
<tr>
<td>14</td>
<td>MODE</td>
<td>0</td>
<td>MODE REG</td>
</tr>
<tr>
<td>15</td>
<td>STAT</td>
<td>MODE+1</td>
<td>STATUS REG</td>
</tr>
<tr>
<td>16</td>
<td>CRDAT</td>
<td>STAT+1</td>
<td>CRT DATA REG</td>
</tr>
<tr>
<td>17</td>
<td>BDAT</td>
<td>KBDAT+1</td>
<td>KB DATA REG</td>
</tr>
<tr>
<td>18</td>
<td>EOL</td>
<td>KBDAT+1</td>
<td>END OF LINE CHAR</td>
</tr>
<tr>
<td>19</td>
<td>BS</td>
<td>EDL+1</td>
<td>BACKSPACE EDIT CHAR</td>
</tr>
<tr>
<td>20</td>
<td>DL</td>
<td>BS+1</td>
<td>DELETE LINE EDIT CHAR</td>
</tr>
<tr>
<td>21</td>
<td>CPDA</td>
<td>2</td>
<td>CP DATA AVAIL FLAG</td>
</tr>
<tr>
<td>22</td>
<td>CRTBSY</td>
<td>1</td>
<td>CRT BUSY FLAG</td>
</tr>
<tr>
<td>23</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>0000</td>
<td>314020</td>
<td>BEGIN:</td>
</tr>
<tr>
<td>25</td>
<td>0003</td>
<td>3E1E</td>
<td>ORG 0</td>
</tr>
<tr>
<td>26</td>
<td>0005</td>
<td>D310</td>
<td>BEGIN:</td>
</tr>
<tr>
<td>27</td>
<td>0007</td>
<td>B11</td>
<td>BEGIN:</td>
</tr>
<tr>
<td>0009</td>
<td>CB5F</td>
<td>33</td>
<td>BEGIN:</td>
</tr>
<tr>
<td>000B</td>
<td>8FPA</td>
<td>34</td>
<td>BEGIN:</td>
</tr>
<tr>
<td>000D</td>
<td>3E00</td>
<td>36</td>
<td>BEGIN:</td>
</tr>
<tr>
<td>000F</td>
<td>D310</td>
<td>37</td>
<td>BEGIN:</td>
</tr>
<tr>
<td>0011</td>
<td>AF</td>
<td>38</td>
<td>BEGIN:</td>
</tr>
<tr>
<td>0012</td>
<td>F602</td>
<td>42</td>
<td>SET KB ENABLE BIT</td>
</tr>
<tr>
<td>0014</td>
<td>F601</td>
<td>44</td>
<td>SET CRT ENABLE BIT</td>
</tr>
<tr>
<td>0016</td>
<td>F604</td>
<td>45</td>
<td>SET BLOCK MOVE BIT</td>
</tr>
<tr>
<td>0018</td>
<td>F608</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>001A</td>
<td>D311</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>001C</td>
<td>3E15</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>001E</td>
<td>D310</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>0020</td>
<td>21B100</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>0022</td>
<td>0E11</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>0025</td>
<td>0008</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>0027</td>
<td>EDB3</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>0029</td>
<td>3E01</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>002B</td>
<td>D310</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>002D</td>
<td>DB11</td>
<td>56</td>
<td></td>
</tr>
</tbody>
</table>

1-105
AND CPDAV

JR Z, LOOP1 ; LOOP UNTIL SET

LD A, KBDAT ; GET BYTE COUNT

OUT (CPORT), A

IN A, (DPORT)

LD B, A ; SAVE IN B

LD D, A ; COPY TO D

SET

LD A, KBDAT ; GET BYTE COUNT

OUT (CPORT), A

LD C, DPORT

LD HL, MSSG+1

INIR

LD A, DIND ; READ DATA LINE

OUT (CPORTl, A

INC B ; ALLOW LF CHAR

CALL CRTOUT ; OUTPUT CRT DATA

LD HL, MSSG

LD C, DPORT

LD A, DIND ; WRITE TO DIND

OUT (DPORT), A

OUT (CPORTl, A

LD A, (HL)

CP '·'

RET Z

LD B, A

CALL CRTOUT

INC HL

JR SO

CALL CRTOUT ; WRITE BLOCK LENGTH

LD A, STAT ; WAIT FOR CRT

OUT (CPORTl, A

DELAY:

IN A, (DPORT)

AND CRTBSY

JR NZ, DELAY

LD A, STAT ; THEN CLEAR CPDAV

OUT (CPORTl, A

LD A, STAT ; THEN CLEAR CPDAV

OUT (CPORTl, A

IN A, (DPORT)

AND OFFH-CPDAV

OUT (DPORT), A

LD B, D ; RESTORE BYTE COUNT

INC B

INC B

CALL CRTOUT ; OUTPUT CRT DATA

LD HL, MSSG

CALL SO

LD B, MSGEND-MSSG

CALL CRTOUT ; WRITE TO DIND

LD A, STAT ; THEN SET CRT BUSY

OUT (CPORTl, A

LD A, (DPORT)

OUT (CPORTl, A

IN A, (DPORT)

LD A, (DPORT)

OUT (CPORTl, A

LD A, (DPORT)

OUT (CPORTl, A

LD A, (DPORT)

OUT (CPORTl, A

JR LOOP

JR LOOP

JR LOOP

JR LOOP

JR LOOP

JR LOOP

JR LOOP

JR LOOP

JR LOOP

JR LOOP

JR LOOP

JR LOOP

JR LOOP

JR LOOP
<table>
<thead>
<tr>
<th>Line</th>
<th>Address</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0092</td>
<td>D311</td>
<td>193 OUT (DPORT), A</td>
</tr>
<tr>
<td>0094</td>
<td>C9</td>
<td>194 RET</td>
</tr>
<tr>
<td>0095</td>
<td>3E01</td>
<td>200 LD A, STAT, READ UPC STATUS</td>
</tr>
<tr>
<td>0097</td>
<td>D310</td>
<td>201 OUT (CPORT), A</td>
</tr>
<tr>
<td>0099</td>
<td>DB11</td>
<td>202 KB11:</td>
</tr>
<tr>
<td>009B</td>
<td>E602</td>
<td>203 IN A, (DPORT), CP DAV?</td>
</tr>
<tr>
<td>009D</td>
<td>2BFA</td>
<td>204 AND CPDAV</td>
</tr>
<tr>
<td>009F</td>
<td>3E03</td>
<td>205 JR Z, KB11 AND LOOP</td>
</tr>
<tr>
<td>00A1</td>
<td>D310</td>
<td>206 LD A, KBDAT ELSE, READ DATA</td>
</tr>
<tr>
<td>00A3</td>
<td>DB11</td>
<td>207 OUT (CPORT), A</td>
</tr>
<tr>
<td>00A5</td>
<td>47</td>
<td>208 IN A, (DPORT)</td>
</tr>
<tr>
<td>00A6</td>
<td>3E01</td>
<td>209 LD B, A</td>
</tr>
<tr>
<td>00A8</td>
<td>DB11</td>
<td>210 LD A, STAT; CLEAR CP DAV</td>
</tr>
<tr>
<td>00AA</td>
<td>D310</td>
<td>211 OUT (CPORT), A</td>
</tr>
<tr>
<td>00AC</td>
<td>E6FD</td>
<td>212 IN A, (DPORT)</td>
</tr>
<tr>
<td>00AE</td>
<td>D311</td>
<td>213 AND OFFH-CPDAV</td>
</tr>
<tr>
<td>00B0</td>
<td>C9</td>
<td>214 OUT (DPORT), A</td>
</tr>
<tr>
<td>00B2</td>
<td>02</td>
<td>215 RET</td>
</tr>
<tr>
<td>00B3</td>
<td>02</td>
<td>218 *L ON</td>
</tr>
<tr>
<td>00B4</td>
<td>04</td>
<td>219 PRI'BLK: DEFB 1 HOME</td>
</tr>
<tr>
<td>00B5</td>
<td>05</td>
<td>220 DEFB 2 FORD</td>
</tr>
<tr>
<td>00B6</td>
<td>06</td>
<td>221 DEFB 3 BACK</td>
</tr>
<tr>
<td>00B7</td>
<td>07</td>
<td>222 DEFB 4 DOWN</td>
</tr>
<tr>
<td>00B8</td>
<td>08</td>
<td>223 DEFB 5 ERASE PAGE</td>
</tr>
<tr>
<td>00B9</td>
<td>0A</td>
<td>224 DEFB 6 RETURN</td>
</tr>
<tr>
<td>00BA</td>
<td>0D</td>
<td>225 DEFB 7 UP</td>
</tr>
<tr>
<td>00BB</td>
<td>54484520</td>
<td>226 DEFB 8 ERASE LINE</td>
</tr>
<tr>
<td>00ED</td>
<td>24</td>
<td>227 PRMEND: EQU *</td>
</tr>
<tr>
<td>00F1</td>
<td>OA</td>
<td>228 MSG: DEFB OAH</td>
</tr>
<tr>
<td>00F2</td>
<td>OD</td>
<td>229 MSG: DEFB ODH</td>
</tr>
<tr>
<td>00F4</td>
<td>54484520</td>
<td>230 MSG: 'THE QUICK BROWN FOX JUMPED OVER THE LA</td>
</tr>
<tr>
<td>00F6</td>
<td>24</td>
<td>231 MSG: TAIL'</td>
</tr>
<tr>
<td>00F7</td>
<td>24</td>
<td>232 MSG: 'THE QUICK BROWN FOX JUMPED OVER THE LA</td>
</tr>
<tr>
<td>00F8</td>
<td>24</td>
<td>233 MSG: TAIL'</td>
</tr>
<tr>
<td>00F9</td>
<td>24</td>
<td>234 MSG: 'THE QUICK BROWN FOX JUMPED OVER THE LA</td>
</tr>
<tr>
<td>00FA</td>
<td>24</td>
<td>235 MSG: TAIL'</td>
</tr>
<tr>
<td>00FB</td>
<td>24</td>
<td>236 MSG: 'THE QUICK BROWN FOX JUMPED OVER THE LA</td>
</tr>
<tr>
<td>00FC</td>
<td>24</td>
<td>237 MSG: TAIL'</td>
</tr>
<tr>
<td>00FD</td>
<td>24</td>
<td>238 MSG: 'THE QUICK BROWN FOX JUMPED OVER THE LA</td>
</tr>
<tr>
<td>00FE</td>
<td>24</td>
<td>239 MSG: TAIL'</td>
</tr>
</tbody>
</table>

1-107
APPENDIX C

Internal UPC Organization

Figure C-1. Port and Data Definitions for UPC

Figure C-2. UPC Status Bytes and Cursor Control Table
<table>
<thead>
<tr>
<th>Register</th>
<th>CPU Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODE</td>
<td>00</td>
</tr>
<tr>
<td>STATUS</td>
<td>01</td>
</tr>
<tr>
<td>CRDAT</td>
<td>02</td>
</tr>
<tr>
<td>KBDAT</td>
<td>03</td>
</tr>
<tr>
<td>EOL</td>
<td>04</td>
</tr>
<tr>
<td>BS</td>
<td>05</td>
</tr>
<tr>
<td>DL</td>
<td>06</td>
</tr>
<tr>
<td>VECT</td>
<td>07</td>
</tr>
<tr>
<td>DATA INDIRECTION</td>
<td>15</td>
</tr>
</tbody>
</table>

Figure C-3. UPC-to-CPU DSC Registers
Z80® 8-Bit Microprocessor Family
INTRODUCTION

With the variety of microprocessors available today, it is often difficult for users to know which one best suits their needs. The choice can be based on a number of factors, such as unit cost, throughput, code density, ease of programming, compatibility, software and hardware support, and availability of second sources.

In high-volume applications (with quantities exceeding 10,000), the cost of parts, especially of memory, is extremely critical. The right microprocessor should be able to interface to low-cost memory components and should be efficient in its use of memory. In other applications where a large software development effort is required, the cost of such an effort may be of more consequence than the cost of parts. Therefore, in software intensive applications, a microprocessor should be evaluated for its ease of programming. In some applications, a particular task must be done very rapidly, or a large number of tasks must be executed in a small amount of time. Some processors perform particular tasks much faster than others, whereas some might not be as fast at a particular task, but are generally faster than others when a large group of tasks is executed. Unfortunately, a user might have to choose a particular processor because it is the only one that can perform a particular task fast enough, even though it may be less memory efficient and more difficult to program than other processors.

This report compares the capabilities of two microprocessors: the 80 and the 6502. Both have many characteristics in common, but they also have a number of very significant differences. These differences will be discussed in detail, and their significance in terms of memory usage, number of lines of code (ease of programming), and execution speed will be measured by a group of benchmark programs.

Ten different benchmark programs are presented here. They represent many tasks commonly performed by microprocessors, yet are short and simple enough for the reader to understand and verify without much effort. The programs have been optimized for each processor.

COMMON CHARACTERISTICS OF THE Z80 AND THE 6502

The 80 and the 6502 are 40-pin microprocessors. The two processors are clearly similar in many respects. They transfer data to and from external components on an 8-bit data bus. Memory is addressed by a 16-bit address bus. Memory is addressed by a 16-bit address bus. Each processor has various registers that are used for specific functions, such as a 16-bit Program Counter, an 8-bit status register, a Stack Pointer, and an accumulator. The 80 and 6502 both have maskable and nonmaskable interrupt capabilities, both have on-chip clocks, and they can both interface to asynchronous as well as synchronous external devices.

DISTINGUISHING CHARACTERISTICS OF THE Z80 AND THE 6502

Table 1 lists the distinguishing features of the 80 and the 6502. At first glance, the 80 appears to have significantly greater resources than the 6502. Each of these resources should be examined to determine their relative importance.

<table>
<thead>
<tr>
<th></th>
<th>Z80</th>
<th>6502</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Number of 8-bit general-purpose registers</td>
<td>14</td>
<td>3</td>
</tr>
<tr>
<td>2. Number of 16-bit general-purpose registers</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>3. Number of functionally distinct instructions</td>
<td>76</td>
<td>29</td>
</tr>
<tr>
<td>4. Number of addressing modes</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>5. Vectored interrupt capability</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>6. Separate I/O addressing space</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>7. Stack space</td>
<td>64K</td>
<td>256</td>
</tr>
<tr>
<td>8. Dynamic memory refresh capability</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

Table 1. Distinguishing Architectural Features
One of the most striking differences between the Z80 and the 6502 is the number of registers each has (Figure 1). Excluding the Program Counter, Stack Pointer, and Status (Flag) register, the Z80 has 14 general-purpose registers and four special-purpose registers, and the 6502 has one accumulator and two index registers.

Registers in the CPU can be accessed much more rapidly than external memory; therefore, the more data that can be kept and manipulated in registers, the faster a program can execute. A program, however, consists of instructions that are located in external memory, and all data must, at one time or another, be transferred to or from external memory. If a CPU could be designed to work rapidly and efficiently with external memory, the importance of a large register set would be diminished.

The most disturbing aspect of the 6502 register set is not the number of registers, but the size of each. All of the programmer accessible registers in the 6502 are eight bits long. This is a problem because the 6502 has 16-bit addressing just like the Z80 has, and without 16-bit registers, the 6502 provides no convenient mechanism for manipulating addresses.

The Z80 can pair its general-purpose 8-bit registers, forming six 16-bit registers in addition to its two 16-bit index registers. The term "index" used to describe the Z80 registers IX and IY is somewhat of a misnomer. The real usefulness of registers IX and IY is in base register addressing. Benchmark program number 10 (See Appendix B) illustrates the use of register IX in accessing specific bytes within a variably located (dynamic) memory block.

The 6502 index registers are very useful in indexing small data structures. Being only 8-bits long, however, the 6502 index registers cannot be used in data structures of more than 256 bytes, except by breaking larger structures down into 256 byte sections (pages), as illustrated in benchmark programs 4, 5 and 9 (see Appendix C).

The 6502 design concentrates on quick and efficient exchanges between registers and external memory. This is evident in the large number of addressing modes. Nearly all of the 6502 instructions can address memory directly (absolute addressing), and many instructions have indexed addressing. A number of 6502 instructions have a special form of pre- and post-indexed indirect addressing as well.
An interesting feature of the 6502 is its Base Page (or Page Zero) Addressing mode. In Base Page Addressing, the upper 8-bits of the 16-bit address are assumed to be zero. This mode is therefore only applicable to the first 256 bytes of memory. The advantage of Base Page Addressing is that only one byte is needed to specify an address. With single-byte addressing, instructions can be shorter in length and therefore can execute faster than instructions containing 16-bit addresses. The base page assumption is also available in the indexed addressing modes. In the pre- and post-indexed indirect addressing modes referred to above, the location of the indirect address is always assumed to be in page zero. Pre-indexed indirect addressing works only with index register X, and post-indexed indirect addressing works only with index register Y. All of these addressing modes are very important and very useful, especially when dealing with the first 256 bytes of memory.

Another interesting characteristic of the 6502 is that its Stack Pointer is only eight bits long. An 8-bit Stack Pointer allows 256 bytes of stack space, which is sufficient for many applications. However, there are applications that require more stack space, and these applications would not be able to use the 6502. The 6502 stack space is dedicated to page one (the second lowest 256 byte area of memory). As with base page addressing, the upper byte of the 16-bit stack address is implied and need not be computed during stack accesses. Instructions in the 6502 that deal with the stack, however, use the Stack Pointer indirectly, so no savings in the length of the address field can be attributed to the stack limitation.

The Z80 has one very important addressing mode not found in the 6502, referred to as Indirect Register Addressing. In this mode, the operand is in a memory location specified by the address residing in a 16-bit register pair. With a 16-bit address, this mode can cover the entire memory space of the Z80. Since the register holding the address is a pair of 8-bit registers, the upper and lower halves can be manipulated independently to access different bytes within a page or the same byte in different pages. Another important quality of Indirect Register Addressing is that instructions using this mode need to specify only the register pair and not the address itself. This allows instructions to be shorter than instructions using other addressing modes.

Addressing modes are not realized without cost. Every instruction a processor has must be represented by an opcode. One of the most fundamental factors affecting the efficiency of a processor is its instruction encoding. It is important to keep instructions as short as possible, because the length of instructions affects the amount of memory used by a program and the program execution time. If the opcode size is held to a fixed length, such as one byte, the number of possible instructions decreases as the number of addressing modes increases. Instructions whose opcodes imply the operands, as in Register and Indirect Register Addressing, need only be one byte long, whereas instructions with other addressing modes, such as Direct, Indirect, Base Page, and Indexed, must further contain the address itself and so are two or three bytes long. A comparison of the Z80 and the 6502 is a perfect example of this point: when operand combinations are considered, the Z80 has 202 different one-byte instructions, and the 6502 has only 29 one-byte instructions (see Table 2).

<table>
<thead>
<tr>
<th>Table 2. Instruction Length Data*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Average number of bytes per instruction</td>
</tr>
<tr>
<td>Number of instructions taking</td>
</tr>
<tr>
<td>1 byte</td>
</tr>
<tr>
<td>2 byte</td>
</tr>
<tr>
<td>3 byte</td>
</tr>
<tr>
<td>4 byte</td>
</tr>
</tbody>
</table>

*Instruction counts here include permutations of operand possibilities including registers and addressing modes but not permutations of memory addresses.

In the Z80, 16-bit registers are useful not only in addressing but also in manipulating 16-bit data. The Z80 provides instructions to add, subtract, increment, decrement, load, store, and exchange 16-bit registers. The 6502 has no 16-bit data manipulation instructions. Manipulating 16-bit data with the 6502 usually requires several more instructions than equivalent operations with the Z80.

The number of instructions a processor has and the usefulness of those instructions are important factors in the number of instructions required to perform a particular task. Other important factors are the addressing modes and the number of accumulators or registers capable of being the destination of arithmetic operations. The more accumulators a processor has, the fewer extraneous instructions are needed to move data to where it can be manipulated. The 6502 has one 8-bit accumulator through which every add and subtract operation must pass. The Z80, on the other hand, has two 8-bit accumulators (A and A') and four 16-bit registers that can be the destination of arithmetic operations (HL, HL*, IX, and IY).

Both the Z80 and the 6502 have interrupts. The Z80 has the additional capability of automatically vectoring to up to 128 different programmable locations when interrupts occur. An 8-bit jump table vector is automatically asserted by Zilog.
Z80 peripherals. Vectoring reduces interrupt response time by eliminating the need for software polling to determine the source of an interrupt in multiple interrupt systems. The Z80 also has non-vectoring interrupt modes for use in less complex systems. The 6502 has no interrupt vectoring capability.

Another important difference between the two CPUs in question is the way they address input and output. The 6502 has no special provisions for I/O addressing and simply interfaces to input and output devices as part of its memory space. This is referred to as memory-mapped I/O. The Z80 has specific I/O instructions and a specific I/O address space of 256 bytes in addition to its memory addressing space. Keeping I/O in a separate addressing space keeps the main memory map clear and reduces the chances of an output device being erroneously written to by runaway programs. If the need for memory-mapped I/O addressing ever arises, the Z80 can accommodate the need in the same manner as the 6502.

Dynamic memory is used in many microprocessor applications. The Z80 can refresh dynamic memory automatically without special refresh circuitry. This feature can reduce the cost of a board by decreasing the number of components needed. The 6502 has no refresh capability. Moreover, it is particularly difficult to interface the 6502 with dynamic RAM because of the critical nature of its memory access timing.

The Z80 and the 6502 are available in various versions, specified by a letter appended to the root name, for example, Z80A or 6502B. The version, in the case of both of these microprocessors is closely related to its memory access timing (see Table 3). Notice that the memory access timing for a Z80A is very close to the memory timing for a 6502A. Notice also that the clock frequency of the Z80A is twice that of the 6502A.

### Table 3. Memory Access Times for Various Clock Rates

<table>
<thead>
<tr>
<th></th>
<th>Memory Access Time</th>
<th>Clock Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z80</td>
<td>575 ns</td>
<td>2.5 MHz</td>
</tr>
<tr>
<td>6502</td>
<td>650 ns</td>
<td>1.0 MHz</td>
</tr>
<tr>
<td>Z80A</td>
<td>325 ns</td>
<td>4.0 MHz</td>
</tr>
<tr>
<td>6502A</td>
<td>310 ns</td>
<td>2.0 MHz</td>
</tr>
<tr>
<td>Z80B</td>
<td>190 ns</td>
<td>6.0 MHz</td>
</tr>
<tr>
<td>6502B</td>
<td>170 ns</td>
<td>3.0 MHz</td>
</tr>
</tbody>
</table>

The memory access timing of a microprocessor is important when evaluating the overall speed and the cost of a particular application. Faster memory components are much more expensive and difficult to obtain than slower ones. The Z80 has a built-in provision for interfacing with components that cannot respond in the normal access time. The Z80 has an input pin called WAIT that can be activated whenever a slow device is addressed. Activating the WAIT input causes the Z80 to add discrete clock cycles to its access timing. The 6502 can interface to slower components by controlling the clock directly, but doing so requires much more critical timing considerations than the method used with the Z80, and it defeats the usefulness of the 6502's internal clock circuitry. Moreover, variations in the main clock might not be tolerable to other devices in the system.

Interfacing the 6502 to program memory that cannot respond at full speed is futile, because 90 percent of the 6502 clock cycles are typically program memory accesses and little would be gained by extending those cycles. It is, however, quite productive to use a high-speed Z80 with program memory that cannot respond at full speed, because, typically, less than 25 percent of the Z80 clock cycles are program memory accesses and extending those cycles would have relatively little effect on overall execution speed.

### BENCHMARK RESULTS

There are so many factors involved in ascertaining a processor's capabilities that it is difficult to determine specific figures without actually writing benchmark programs. When evaluating a processor for use in a particular application, the user should use programs representative of his or her application. This report is intended for a general audience of users and presents a wide variety of program types (see Appendix A for the benchmark program specifications).

Three different aspects of performance are measured by the benchmark programs here:

1. Memory utilization
2. Ease of Programming
3. Execution Speed

Memory utilization is often the most important criterion in measuring the performance of a processor. It measures the amount of memory (usually program memory) used by the processor in performing various tasks. It is important, because the cost of memory is often one of the dominating costs of a microprocessor application. Table 4 lists the number of bytes of program memory used by the Z80 and the 6502 in each of the benchmark programs.

The ease of programming is a somewhat subjective issue, but very important nonetheless. Software development costs are enormous and can outweigh many other considerations made by microprocessor users. One measure of the ease of programming is the number of instructions (lines of code) required to perform a given task. This measure is used in this report because of its simplicity and objectivity. The number of lines of source code in the benchmark programs for each of the microprocessors is shown in Table 5.
### Table 4. Number of Bytes of Program Memory Used

<table>
<thead>
<tr>
<th>Program Description</th>
<th>Z80</th>
<th>6502</th>
<th>Ratio 6502/Z80</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computed GOTO Implementation</td>
<td>9</td>
<td>27</td>
<td>3.00</td>
</tr>
<tr>
<td>8 x 8 Bit Multiply Routine</td>
<td>26</td>
<td>41</td>
<td>1.58</td>
</tr>
<tr>
<td>16 x 16 Bit Multiply</td>
<td>20</td>
<td>44</td>
<td>2.20</td>
</tr>
<tr>
<td>Block Move</td>
<td>11</td>
<td>51</td>
<td>4.64</td>
</tr>
<tr>
<td>Linear Search</td>
<td>8</td>
<td>41</td>
<td>5.13</td>
</tr>
<tr>
<td>Insert into Linked List</td>
<td>12</td>
<td>19</td>
<td>1.58</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>23</td>
<td>31</td>
<td>1.35</td>
</tr>
<tr>
<td>Interrupt Handling</td>
<td>6</td>
<td>11</td>
<td>1.83</td>
</tr>
<tr>
<td>Character String Translation</td>
<td>17</td>
<td>48</td>
<td>2.82</td>
</tr>
<tr>
<td>Dynamic Memory Access</td>
<td>11</td>
<td>24</td>
<td>2.18</td>
</tr>
<tr>
<td><strong>Average ratio 6502/Z80</strong></td>
<td></td>
<td></td>
<td>2.63</td>
</tr>
</tbody>
</table>

### Table 5. Number of Lines of Source Code

<table>
<thead>
<tr>
<th>Program Description</th>
<th>Z80</th>
<th>6502</th>
<th>Ratio 6502/Z80</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computed GOTO Implementation</td>
<td>8</td>
<td>17</td>
<td>2.13</td>
</tr>
<tr>
<td>8 x 8 Bit Multiply Routine</td>
<td>14</td>
<td>20</td>
<td>1.43</td>
</tr>
<tr>
<td>16 x 16 Bit Multiply</td>
<td>11</td>
<td>23</td>
<td>2.09</td>
</tr>
<tr>
<td>Block Move</td>
<td>4</td>
<td>27</td>
<td>6.75</td>
</tr>
<tr>
<td>Linear Search</td>
<td>3</td>
<td>22</td>
<td>7.33</td>
</tr>
<tr>
<td>Insert into Linked List</td>
<td>6</td>
<td>10</td>
<td>1.67</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>15</td>
<td>15</td>
<td>1.00</td>
</tr>
<tr>
<td>Interrupt Handling</td>
<td>6</td>
<td>7</td>
<td>1.17</td>
</tr>
<tr>
<td>Character String Translation</td>
<td>10</td>
<td>26</td>
<td>2.60</td>
</tr>
<tr>
<td>Dynamic Memory Access</td>
<td>3</td>
<td>13</td>
<td>4.33</td>
</tr>
<tr>
<td><strong>Average ratio 6502/Z80</strong></td>
<td></td>
<td></td>
<td>3.05</td>
</tr>
</tbody>
</table>

### Table 6. Program Execution Times for the Lowest Speed Versions

<table>
<thead>
<tr>
<th>Program Description</th>
<th>usec Z80</th>
<th>usec 6502</th>
<th>Ratio 6502/Z80</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computed GOTO Implementation</td>
<td>20.27</td>
<td>46.33</td>
<td>2.29</td>
</tr>
<tr>
<td>8 x 8 Bit Multiply Routine</td>
<td>160.80</td>
<td>196.00</td>
<td>1.22</td>
</tr>
<tr>
<td>16 x 16 Bit Multiply</td>
<td>405.20</td>
<td>713.00</td>
<td>1.76</td>
</tr>
<tr>
<td>Block Move</td>
<td>16138.00</td>
<td>31816.00</td>
<td>1.97</td>
</tr>
<tr>
<td>Linear Search</td>
<td>8406.00</td>
<td>13011.00</td>
<td>1.55</td>
</tr>
<tr>
<td>Insert into Linked List</td>
<td>24.80</td>
<td>34.00</td>
<td>1.37</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>250718.00</td>
<td>280474.00</td>
<td>1.12</td>
</tr>
<tr>
<td>Interrupt Handling</td>
<td>17.2</td>
<td>32.00</td>
<td>1.86</td>
</tr>
<tr>
<td>Dynamic Memory Access</td>
<td>27.60</td>
<td>47.00</td>
<td>1.70</td>
</tr>
<tr>
<td><strong>Average ratio 6502/Z80</strong></td>
<td></td>
<td></td>
<td>1.65</td>
</tr>
</tbody>
</table>

* Z80 maximum clock frequency is 2.5 MHz. Memory access time is 575 ns.
* 6502 maximum clock frequency is 1.0 MHz. Memory access time is 650 ns.
Execution speed can be important in several ways. A computer product that has a human interface, such as a keyboard and display, will be more productive and enjoyable to use if it responds quickly. A microprocessor being evaluated for use in controlling a high-speed device might have to be rejected if it cannot meet very rigid timing requirements.

Execution time varies significantly depending on which version of Z80 or 6502 is used, so a comparison of different versions is important. Table 6 lists the execution times of the benchmark programs for the lowest speed versions of the two microprocessors.

The most relevant comparison of execution times is shown in Table 7, where the data is calculated from versions of the Z80 and 6502 that can operate in systems of similar speeds. One should not be confused by the higher clock rate of the Z80B, because even at twice the clock rate of the 6502B, the Z80B has a longer external component access time than the 6502B (see Table 3).

**CONCLUSION**

The results of the benchmark programs presented in this report show the Z80 performing significantly better than the 6502 in nearly every aspect. In six of the ten programs, the 6502 used more than twice the amount of program memory than the Z80. In the bubble sort program, the 6502's best relative performance, it used 35 percent more program memory than the Z80. The number of lines of code used varies dramatically from one program to another, but none of the programs have fewer lines of 6502 code than Z80 code. Comparing versions of equivalent speed (Table 7), the Z80 executes eight of the ten programs in less time than the 6502.

In all three measures of performance (Tables 4, 5, and 7), the program that yields the best results for the 6502 is the bubble sort. The bubble sort program, as specified in Appendix A, operates on an array of less than 256 bytes, so one of the 8-bit index registers in the 6502 can be used very effectively. In applications that primarily use short byte-oriented data structures, the 6502 is worthy of consideration.

Some of the benchmark programs reveal outstanding results in favor of the Z80. For example, the linear search program and the dynamic memory block access program have only three Z80 instructions, and the block move program uses only eight bytes of program memory. The reason for such outstanding results with the Z80 is that it has many exceedingly powerful instructions. The Block Move and Block Search instructions illustrated in the benchmark programs are only a subset of the many block-oriented instructions of the Z80. The ability to access and manipulate bytes in dynamic memory blocks spans nearly the entire Z80 instruction set and is greatly appreciated by programmers who deal with multi-tasking software.

In applications that require data structures longer than 256 bytes or that manipulate 16-bit data, the Z80 is likely to be more efficient than the 6502, particularly in terms of memory utilization and programmer productivity.

### Table 7. Execution Times for Versions with Equivalent Memory Access Time*

<table>
<thead>
<tr>
<th>Program Description</th>
<th>usec Z80B</th>
<th>usec 6502B</th>
<th>Ratio 6502B/Z80B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computed GOTO Implementation</td>
<td>8.45</td>
<td>15.44</td>
<td>1.83</td>
</tr>
<tr>
<td>8 x 8 Bit Multiply Routine</td>
<td>67.00</td>
<td>65.33</td>
<td>0.98</td>
</tr>
<tr>
<td>16 x 16 Bit Multiply</td>
<td>168.83</td>
<td>237.67</td>
<td>1.41</td>
</tr>
<tr>
<td>Block Move</td>
<td>6724.17</td>
<td>10605.33</td>
<td>1.58</td>
</tr>
<tr>
<td>Linear Search</td>
<td>3502.50</td>
<td>4337.00</td>
<td>1.24</td>
</tr>
<tr>
<td>Insert into Linked List</td>
<td>10.33</td>
<td>11.33</td>
<td>1.10</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>104465.83</td>
<td>93491.33</td>
<td>0.89</td>
</tr>
<tr>
<td>Interrupt Handling</td>
<td>7.17</td>
<td>10.67</td>
<td>1.49</td>
</tr>
<tr>
<td>Character String Translation</td>
<td>5678.33</td>
<td>7356.00</td>
<td>1.30</td>
</tr>
<tr>
<td>Dynamic Memory Access</td>
<td>11.50</td>
<td>15.67</td>
<td>1.36</td>
</tr>
<tr>
<td>Average ratio 6502B/Z80B</td>
<td></td>
<td></td>
<td>1.32</td>
</tr>
</tbody>
</table>

* Z80B maximum frequency is 6 MHz. Memory access time is 190 ns.
* 6502B maximum clock frequency is 3 MHz. Memory access time is 170 ns.
APPENDIX A. BENCHMARK PROGRAM SPECIFICATION

Computed GOTO implementation. A byte is tested for three states: negative, zero, and positive. The processor branches to a different variable address for each state.

The byte is in a register, and the three 16-bit addresses are on the stack.

8 x 8 Bit Unsigned Multiply Routine. Two 8-bit unsigned integers (INT1, INT2) located randomly in memory (RAM or ROM) are multiplied together to form a 16-bit product (INT3) to be stored in RAM.

16 x 16 Bit Unsigned Multiply. Two 16-bit unsigned integers, located wherever is most efficient, are multiplied together to form a 32-bit product.

Block Move. Move a block of memory from one location to another. The source and destination addresses and the block size are known at assembly time, but no restriction on their values are allowed.

Use a block size of 1920 bytes (a typical CRT screen) for time calculation.

Linear Search. Search for the first occurrence of a certain byte in a string of bytes. The string address and length are known at assembly time, but no restrictions on their values are allowed.

Use string length equal to 1000 with no find for time calculations.

Insert into Linked List. The linked list exists in RAM (not page zero) and has 160 bit forward pointers. The root (pointer to top entry) may be in page zero.

The address of the entry to be inserted is specified wherever is most efficient. Insert the entry into the top position.

Bubble sort. Using a standard bubble sorting algorithm, arrange an array of bytes (length 256) into descending order.

To calculate the timing, use a length of 100 and assume that the array is in ascending order before sorting.

Interrupt Handling. Respond to an interrupt, save processor status, save registers, restore registers, restore processor status, and return.

Response time does not include the time for an executing instruction to complete.

Character String Translation. A string of ASCII characters of known length is translated into EBCDIC according to an existing 256 byte translation table.

Use a length of 1000 for time calculations.

Dynamic Memory Access. The following operations are performed on bytes within a 256 byte dynamic memory block (dynamic means the block address is a variable).

Set bit 5 of byte 151, increment byte 70, and shift byte 205 left.
APPENDIX B: Z80 PROGRAM LISTINGS

1. Z80 Computed GOTO implementation

```
! COMPUTED GOTO (REG A CONTAINS THE BYTE TO BE TESTED)

<table>
<thead>
<tr>
<th>bytes</th>
<th>cycles</th>
<th>!</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>C GO TO</td>
<td>POP DE</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>POP HL</td>
<td>OR A</td>
</tr>
<tr>
<td>1</td>
<td>11/5</td>
<td>RET M</td>
<td>POP BC</td>
</tr>
<tr>
<td>2</td>
<td>12/7</td>
<td>JR Z,C GO TO</td>
<td>EX DE, HL</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>C GO TO</td>
<td>JP (HL)</td>
</tr>
</tbody>
</table>

Lines = 8
Bytes = 9
Cycles = 50.67
```

2. Z80 8 x 8 Bit Unsigned Multiply Routine

```
! PREPARE ARGUMENTS FOR SUBROUTINE

<table>
<thead>
<tr>
<th>bytes</th>
<th>cycles</th>
<th>!</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>13</td>
<td>LD A,(INT1)</td>
<td>!RANDOM LOCATION</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>LD E,A</td>
<td>!REG E = MULTIPLICAND</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td>LD A,(INT2)</td>
<td>!REG A = MULTIPLIER</td>
</tr>
<tr>
<td>3</td>
<td>17</td>
<td>CALL MULT8</td>
<td>!CALL SUBROUTINE</td>
</tr>
</tbody>
</table>

! 8 X 8 UNSIGNED MULTIPLY ROUTINE

<table>
<thead>
<tr>
<th>bytes</th>
<th>cycles</th>
<th>!</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
<td>MULT8 LD D,0</td>
<td>!EXTEND MULTIPLICAND TO 16 BIT</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>LD H,D</td>
<td>!INITIALIZE MULTIPLIER/PRODUCT</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>LD L,A</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>LD B,8</td>
<td>!INITIALIZE LOOP COUNTER</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>MULTI10 ADD HL,HL</td>
<td>!SHIFT MULTIPLIER/PRODUCT LEFT</td>
</tr>
<tr>
<td>2</td>
<td>12/7</td>
<td>JR NC,MULT20</td>
<td>!JUMP IF MSB OF MULTIPLIER WAS 0</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>ADD HL,DE</td>
<td>!ADD MPCAND TO PRODUCT</td>
</tr>
<tr>
<td>2</td>
<td>13/8</td>
<td>MULT20 DJNZ MULTI10</td>
<td>!DEC LOOP CNTR &amp; JMP IF NOT 0</td>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td>RET</td>
<td>!RETURN</td>
</tr>
</tbody>
</table>

! STORE PRODUCT

<table>
<thead>
<tr>
<th>bytes</th>
<th>cycles</th>
<th>!</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>16</td>
<td>LD (INT3),HL</td>
</tr>
</tbody>
</table>

Lines = 14
Bytes = 26
Cycles = 402 average
### 3. Z80 16 x 16 Bit Unsigned Multiply

- **! 16 x 16 BIT UNSIGNED MULTIPLY**
- **! BC = MULTIPLICAND**
- **! DE = MULTIPLIER / PRODUCT MSW**

<table>
<thead>
<tr>
<th>bytes</th>
<th>cycles</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
<td>MULT16</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>LD A,16 !A = LOOP COUNT</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>LD BC</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>ADD HL,HL !SHIFT MULTIPLIER/PRODUCT LEFT</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>RL D</td>
</tr>
<tr>
<td>2</td>
<td>12/7</td>
<td>JR NC,MULT30</td>
</tr>
<tr>
<td>1</td>
<td>11</td>
<td>ADD HL,BC</td>
</tr>
<tr>
<td>2</td>
<td>12/7</td>
<td>JR NC,MULT40</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>INC DE</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>MULT40 DEC A</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>JP NZ,MULT30</td>
</tr>
</tbody>
</table>

Lines = 11  
Bytes = 20  
Cycles = 1013 average

### 4. Z80 Block Move

- **! Move a block of memory.**

<table>
<thead>
<tr>
<th>bytes</th>
<th>cycles</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>10</td>
<td>BLKMOV</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>LD DE,DESTIN</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>LD BC,BLKSIZ</td>
</tr>
<tr>
<td>2</td>
<td>21/16</td>
<td>LDIR</td>
</tr>
</tbody>
</table>

Lines = 4  
Bytes = 11  
Cycles = 40345

### 5. Z80 Linear Search

- **! SEARCH FOR THE BYTE IN REG A**

<table>
<thead>
<tr>
<th>bytes</th>
<th>cycles</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>10</td>
<td>SEARCH</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>LD BC,LENGTH</td>
</tr>
<tr>
<td>2</td>
<td>21/16</td>
<td>CPIR</td>
</tr>
</tbody>
</table>

Lines = 3  
Bytes = 8  
Cycles = 21015
6. Z80 Insert into a Linked List

<table>
<thead>
<tr>
<th>byte</th>
<th>cycles</th>
<th>!</th>
<th>INSERT THE ENTRY POINTED TO BY (HL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>13</td>
<td>INSERT</td>
<td>LD A,(ROOT) !XFER OLD TOP ENTRY PTR</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td></td>
<td>LD (HL),A</td>
</tr>
<tr>
<td>3</td>
<td>13</td>
<td></td>
<td>LD A,(ROOT+1)</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td></td>
<td>LD (ROOT),HL !ROOT POINTS TO NEW ENTRY</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>INC HL</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>LD (HL),A</td>
<td></td>
</tr>
</tbody>
</table>

Lines = 6
Bytes = 12
Cycles = 62

7. Z80 Bubble Sort

<table>
<thead>
<tr>
<th>byte</th>
<th>cycles</th>
<th>!</th>
<th>BUBBLE SORT ARRAY INTO DESCENDING ORDER</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>10</td>
<td>SORT</td>
<td>LD HL,ARRAY !INIT ARRAY POINTER</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td></td>
<td>LD BC,PAIRCT*256 !INIT PAIR CNTR &amp; EXCHANGE FLAG</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>SORT20</td>
<td>LD A,(HL) !GET FIRST BYTE OF PAIR</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>INC HL !ADDRESS NEXT BYTE</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>LD E,(HL) !GET SECOND BYTE OF PAIR</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>CP E !COMPARE FIRST &amp; SECOND BYTE</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>12/7</td>
<td>JR NC,SORT30 !JUMP IF FIRST &gt;= SECOND</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>LD C,1 !SET EXCHANGE FLAG</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>LD (HL),A !EXCHANGE THE PAIR</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>DEC HL</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>LD (HL),E</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>INC HL</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>13/8</td>
<td>SORT30</td>
<td>DJNZ SORT20 !LOOP TILL ALL PAIRS EXAMINED</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>DEC C !CHECK EXCHANGE FLAG</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>12/7</td>
<td>END !JUMP IF EXCHANGE OCCURRED</td>
<td></td>
</tr>
</tbody>
</table>

Lines = 15
Bytes = 23
Cycles = 626795
8. Z80 Interrupt Handling

```
bytes  cycles
---  ----
1 4    INTRPT EX AF,AF' !SAVE REGISTERS AND STATUS
1 4    EXX
1 4    EXX !RESTORE REGISTERS AND STATUS
1 4    EX AF,AF'
1 4    EI
1 10   RET !RETURN TO INTERRUPTED PROGRAM

Lines = 6
Bytes = 6
Cycles = 43
```

9. Z80 Character String Translation

```
bytes  cycles
---  ----
3 10   TRANSL LD HL,STRING !HL = STRING ADDRESS
2 7    LD D,HI TABLE !D = HIGH BYTE OF XLATION TABLE
2 7    LD B,LO LENGTH !B = LOOP COUNTER LOW BYTE
2 7    LD C,HI LENGTH+1 !C = LOOP COUNTER HIGH BYTE
1 7    TRAN10 LD E,(HL) !GET AN ASCII CHARACTER
1 7    LD A,(DE) !USE IT TO INDEX EBCDIC TABLE
1 7    LD (HL),A !STORE EBCDIC CHAR IN STRING
2 13/8 DJNZ TRAN10 !DEC AND TEST LOOP COUNT
1 4    DEC C
2 12/7 JR NZ1TRAN10 !JUMP IF NOT DONE

Lines = 10
Bytes = 17
Cycles = 34070
```

10. Z80 Dynamic Memory Access

```
bytes  cycles
---  ----
4 23   DYNACC SET 5,(IX+151) !SET BIT 5 OF BYTE 151
3 23   INC (IX+70) !INCREMENT BYTE 70
4 23   SLA (IX+205) !SHIFT BYTE 205 LEFT

DONE END

Lines = 3
Bytes = 11
Cycles = 69
```
1. 6502 Computed GOTO implementation

<table>
<thead>
<tr>
<th>bytes</th>
<th>cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3/2</td>
<td>3/2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3/2</td>
<td>3/2</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

Lines = 17
Bytes = 27
Cycles = 46.33 average
2. 6502 8 x 8 Bit Unsigned Multiply Routine

! PREPARE ARGUMENTS FOR SUBROUTINE !

<table>
<thead>
<tr>
<th>bytes</th>
<th>cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

LDA INT1 !RANDOM LOCATION
STA MPCAND !PAGE ZERO
LDA INT2 !RANDOM LOCATION
STA MPLIER !PAGE ZERO
JSR MULTB !CALL SUBROUTINE

! 8 X 8 UNSIGNED MULTIPLY ROUTINE !

MULTB LDA 110 !CLEAR LOW BYTE OF PRODUCT
LDX HB !INIT LOOP COUNTER
MULT10 ASL A !SHIFT MULTIPLIER/PRODUCT LEFT
INC MPLIER !DECREMENT LOOP COUNTER
RTS !RETURN

ADD MULTIPLICAND TO PRODUCT

CLC
ADC MPCAND
BCC MULT20 !HANDLE CARRY TO HIGH BYTE
INC MPLIER
DEX
BNE MULT10 !BRANCH IF NOT DONE
RTS !RETURN

STORE PRODUCT

STA INT3 !LOW BYTE
LDA MPLIER !HIGH BYTE
STA INT3+1
END

Lines = 20
Bytes = 41
Cycles = 196 average
3. 6502 16 x 16 Bit Unsigned Multiply

! 16 x 16 UNSIGNED MULTIPLY!

! MPCAND : 2 CONSECUTIVE BYTES IN PAGE 0
! MPLIER : 2 CONSECUTIVE BYTES IN PAGE 0 (PRODUC+2)
! PRODUC : 4 CONSECUTIVE BYTES IN PAGE 0 (OVERLAPPING MPLIER)

<table>
<thead>
<tr>
<th>bytes</th>
<th>cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>3/2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3/2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3/2</td>
</tr>
</tbody>
</table>

LDX #16 !INIT LOOP COUNTER
LDA #0 !INIT PRODUCT LSW
STA PRODUC
STA PRODUC+1

MUL30 ASL PRODUC !SHIFT MULTIPLIER/PRODUCT LEFT
ROL PRODUC+1
ROL MPLIER
ROL MPLIER+1

BCC MUL40 !JUMP IF MSB WAS 0
CLC !MULTIPLICAND+PRODUCT LSW

LDA PRODUC
ADC MPCAND
STA PRODUC
LDA PRODUC+1
ADC MPCAND+1
STA PRODUC+1
LDA PRODUC+2 !PROPAGATE CARRY
ADC #0
STA PRODUC+2
BCC MUL40

INC PRODUC+3
MUL40 DEX !DEC LOOP COUNT

BNE MUL30 !LOOP TILL DONE
END

Lines = 23
Bytes = 44
Cycles = 713 average
4. 6502 Block Move

<table>
<thead>
<tr>
<th>bytes</th>
<th>cycles</th>
<th>! Move a block of memory.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>BLKMOV</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>LDA $0 SOURCE</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>LDA $0 SOURCE</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>STA SRCADR</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>LDA $0 DESTIN</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>STA DSTADR</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>LDA $0 DESTIN</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>STA DSTADR+1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>LDX $0 COUNT</td>
</tr>
<tr>
<td>2</td>
<td>3/2</td>
<td>BEQ LSTPAG</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>LDY 0</td>
</tr>
<tr>
<td>2</td>
<td>5/6</td>
<td>LOOP1 LDA (SCRCADR),Y</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>STA (DSTADR),Y</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>DEY</td>
</tr>
<tr>
<td>2</td>
<td>3/2</td>
<td>BNE LOOP1</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>INC SRCADR+1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>DEX</td>
</tr>
<tr>
<td>2</td>
<td>3/2</td>
<td>BNE LOOP1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>LSTPAG</td>
</tr>
<tr>
<td>2</td>
<td>3/2</td>
<td>BEQ DONE</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>DEC SRCADR</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>DEC DSTADR</td>
</tr>
<tr>
<td>2</td>
<td>5/6</td>
<td>LOOP2 LDA (SCRCADR),Y</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>STA (DSTADR),Y</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>DEY</td>
</tr>
<tr>
<td>2</td>
<td>3/2</td>
<td>BNE LOOP2</td>
</tr>
</tbody>
</table>

DONE END

Lines = 27
Bytes = 51
Cycles = 31B16
5. 6502 Linear Search

<table>
<thead>
<tr>
<th>bytes</th>
<th>cycles</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>SEARCH LDA #LO STRING STA STRADR</td>
<td>SEARCH FOR BYTE IN REG A</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>LDA #HI STRING STA STRADR+1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>LDX #HI COUNT BEQ SRCH20</td>
<td>!X = HIGH BYTE OF COUNT</td>
</tr>
<tr>
<td>2</td>
<td>3/2</td>
<td>LDY #0 BEQ SRCH10 CMP (STRADR),Y</td>
<td>!Y = COUNTER AND INDEX</td>
</tr>
<tr>
<td>2</td>
<td>5/6</td>
<td>SRCH10 CMP (STRADR),Y BEQ FOUND</td>
<td>!MATCH?</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>INY INC STRADR</td>
<td>!UPDATE POINTER TO NEXT 256</td>
</tr>
<tr>
<td>2</td>
<td>3/2</td>
<td>DEX BEQ SRCH20</td>
<td>!DECREMENT HIGH BYTE OF COUNT</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>SRCH20 LDY #LO COUNT CMP (STRADR),Y</td>
<td>!BRANCH IF NOT LAST PAGE</td>
</tr>
<tr>
<td>2</td>
<td>3/2</td>
<td>BEQ DONE</td>
<td>!BRANCH IF NO PARTIAL PAGE</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>LDY #0 CPY #LO COUNT</td>
<td>!Y = INDEX</td>
</tr>
<tr>
<td>2</td>
<td>5/6</td>
<td>SRCH30 CMP (STRADR),Y BEQ FOUND</td>
<td>!DONE WITH LAST PARTIAL PAGE</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>INY</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>CPY #LO COUNT BNE SRCH30</td>
<td>!BRANCH IF NOT</td>
</tr>
<tr>
<td>2</td>
<td>3/2</td>
<td>BNE SRCH30</td>
<td>END</td>
</tr>
</tbody>
</table>

Lines = 22
Bytes = 41
Cycles = 13011

6. 6502 Insert into Linked List

<table>
<thead>
<tr>
<th>bytes</th>
<th>cycles</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>INSERT LDY #0</td>
<td>INSERT THE ENTRY POINTED TO BY (NEWADR)</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>LDA ROOT STA (NEWADR),Y</td>
<td>!FIRST 2 BYTES IS FORWARD PTR</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>LDA ROOT+1 INY</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>STA (NEWADR),Y LDA NEWADR</td>
<td>!ROOT POINTS TO NEW ENTRY</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>STA ROOT LDA NEWADR+1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>STA ROOT+1</td>
<td>END</td>
</tr>
</tbody>
</table>

Lines = 10
Bytes = 19
Cycles = 34
7. Bubble Sort

! BUBBLE SORT ARRAY INTO DESCENDING ORDER
!

<table>
<thead>
<tr>
<th>bytes</th>
<th>cycles</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>SORT</td>
<td>LDY #0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>LDX</td>
<td>#LENGTH-1</td>
</tr>
<tr>
<td>4/5</td>
<td>4/5</td>
<td>LDA</td>
<td>ARRAY,X</td>
</tr>
<tr>
<td>3</td>
<td>3/2</td>
<td>CMP</td>
<td>ARRAY+1,X</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>LDX</td>
<td>ARRAY+1,X</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>PHA</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4/5</td>
<td>LDA</td>
<td>ARRAY+1,X</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>STA</td>
<td>ARRAY,X</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>PLA</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>STA</td>
<td>ARRAY+1,X</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>DEX</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3/2</td>
<td>BNE</td>
<td>SORT10</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>DEY</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3/2</td>
<td>BEQ</td>
<td>SORT</td>
</tr>
</tbody>
</table>

Lines = 15
Bytes = 31
Cycles = 280474

8. 6502 Interrupt Handling

! INTERRUPT OVERHEAD (ADD 7 CYCLES RESPONSE TIME)
!

<table>
<thead>
<tr>
<th>bytes</th>
<th>cycles</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>INTRPT</td>
<td>PHA</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>STX</td>
<td>XSAVE</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>STY</td>
<td>YSAVE</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>LDY</td>
<td>YSAVE</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>LDX</td>
<td>XSAVE</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>PLA</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>RTI</td>
<td></td>
</tr>
</tbody>
</table>

Lines = 7
Bytes = 11
Cycles = 32
9. 6502 Character String Translation

<table>
<thead>
<tr>
<th>bytes</th>
<th>cycles</th>
<th>! TRANSLATE STRING FROM ASCII TO EBCDIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>LDA #$0 STRING !SET UP STRING POINTER</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>STA STRADR</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>LDA #$HI STRING</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>STA STRADR+1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>LDA #$HI LENGTH !CHECK HIGH BYTE OF LENGTH</td>
</tr>
<tr>
<td>2</td>
<td>3/2</td>
<td>BEQ TRAN20 !BRANCH IF STRING &lt; 256 CHARS</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>STA COUNT !INIT COUNT</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>LDY #0 !Y = INDEX FOR PARTIAL STRING</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>TRAN10 LDA (STRADR),Y !TRANSLATE A BYTE</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>TAX</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>LDA TABLE,X</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>STA (STRADR),Y</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>INY !INCREMENT INDEX</td>
</tr>
<tr>
<td>2</td>
<td>3/2</td>
<td>BNE TRAN10 !BRANCH IF NOT DONE WITH PAGE</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>INC STRADR+1 !UPDATE POINTER TO NEXT PAGE</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>DEC COUNT !DECREMENT COUNT</td>
</tr>
<tr>
<td>2</td>
<td>3/2</td>
<td>BNE TRAN10 !BRANCH IF NOT LAST PAGE</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>LDY #$LO COUNT !Y = INDEX/COUNT FOR LAST PAGE</td>
</tr>
<tr>
<td>2</td>
<td>3/2</td>
<td>BEQ DONE !BRANCH IF NO PARTIAL PAGE</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>DEC STRADR !ADJUST POINTER</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>TRAN30 LDA (STRADR),Y !TRANSLATE LAST PARTIAL PAGE</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>TAX</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>LDA TABLE,X</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>STA (STRADR),Y</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>DEY</td>
</tr>
<tr>
<td>2</td>
<td>3/2</td>
<td>BNE TRAN30</td>
</tr>
</tbody>
</table>

DONE BNE TRAN30

Lines = 26
Bytes = 48
Cycles = 22068
10. 6502 Dynamic Memory Access

bytes    cycles    (BLOCK) = ADDRESS OF MEMORY BLOCK

2   2    DYNACC  LDY  #151    !SET BIT 5 OF BYTE 151
2   5    LDA  (BLOCK),Y
2   2    ORA  #20
2   6    STA  (BLOCK),Y
2   2    LDY  #70    !INCREMENT BYTE 70
2   5    LDA  (BLOCK),Y
1   2    CLC
2   2    ADC  #1
2   6    STA  (BLOCK),Y
2   2    LDY  #205   !SHIFT BYTE 205 LEFT
2   5    LDA  (BLOCK),Y
1   2    ASL  A
2   6    STA  (BLOCK),Y

DONE  END

Lines = 13
Bytes = 24
Cycles = 47
The new generation of 16-bit microprocessors allows the system designer to implement a powerful, but cost-effective computer system using the currently available 8-bit peripheral support devices. These processors offer advance block transfer operations that allow blocks of data to be moved between memory and an Input/Output (I/O) device. Although the data transfer rates achieved are very high, they are still inadequate for interfacing some system peripherals such as the new 8" Winchester disk drives. To incorporate such high-speed peripheral devices, the system designer needs to integrate a Direct Memory Access (DMA) controller device into the system. This article illustrates the increase in throughput obtained by integrating an 8-bit DMA device into a 16-bit microprocessor system and discusses the various interface techniques and trade-offs involved in such a task.

Z80 DIRECT MEMORY ACCESS CONTROLLER

A DMA device performs the dedicated task of moving data in a microprocessor system independently of the Central Processing Unit (CPU). The transfers are usually between memory and an I/O device, but some DMAs are capable of moving data from memory to memory or between two I/O devices. In a small microprocessor system, the CPU can normally do these transfers via software, but this results in a reduction of system throughput and ties up the CPU for long periods of time when a large amount of data is to be moved. The response time of the CPU in these CPU-managed transfers is inherently slow and may not be adequate in situations where the nature of data transfers demands fast response. The addition of a DMA device to an 8-bit microprocessor system is easily accomplished, since most 8-bit CPU families have a DMA controller device that shares common family interface protocol. Integrating a DMA device into a 16-bit system poses two options to the system designer. Since 16-bit LSI DMA devices are not presently available, the designer can use the 8-bit devices with additional hardware, or can opt for implementing DMA functions using discrete TTL logic. The latter approach offers the advantage of implementing only those functions that are needed. However, even in the most simple cases, a high part count is required to add DMA capability using this approach. The 8-bit devices, on the other hand, offer extensive, integrated capabilities and require relatively little additional logic to interface to 16-bit processors.

The Z80 DMA is a powerful 8-bit DMA device and, unlike most other DMAs, it takes complete control of the system bus during the data transfer. It generates all bus signals normally generated by the Z80 CPU during a data transfer without any external TTL packages. Data transfers can be accomplished in three different modes. In the Byte mode, one byte of data is transferred at a time, giving control of the system bus to the CPU after each byte transfer. In the Burst mode, a block of data bytes is transferred and data transfer operations continue until the READY signal (normally from an I/O device) becomes inactive. At this time, bus control is returned to the CPU and when the I/O device is ready to move more data (activating the READY signal), the data transfer operation is started again. These bursts of data transfers continue until the whole block has been moved. The Continuous mode operates in the same fashion as the Burst mode, except that the bus control is returned to the CPU only when the operation is complete. If the READY signal goes inactive before the whole block is moved, the DMA simply pauses until it becomes active again. In addition to data transfers, the Z80 DMA can also search for a specific data byte. In the Search mode, data bytes are compared to a programmable "match byte" and an interrupt may be generated when a match is found.

The Z80 DMA can generate two port addresses, with either address being variable or fixed. It is capable of doing a data transfer from memory to memory or between two I/O devices, using a single channel in any of the three
COMPARISON OF RATES IN A SMALL SYSTEM

Table 1 illustrates the various transfer speeds that can be obtained in a microprocessor system with a Z80A CPU, a Z8000 CPU, or a Z80A DMA. The Z80A DMA can achieve an impressive transfer rate of 1 Mbyte/sec. The Z80A CPU, using the powerful block transfer instruction, can transfer data at 0.19 Mbytes/sec. Since the DMA achieves the 1 Mbyte/sec transfer rate using two-clock-cycle operations for each byte of transferred data, it requires memory devices with relatively short access times. The Z8000 CPU has a maximum memory-to-memory data transfer rate of 0.44 Mtransfers/sec, and a maximum 1/O-to-memory data transfer rate of 0.40 Mtransfers/sec. The same transfer rates are obtained by the Z8000 CPU whether the data transferred is a byte or a word. However, since the DMA can be made to transfer words with some additional hardware, it can still provide a data transfer rate of 1 Mtransfer/sec. In addition, the DMA can also be programmed to search for a specific byte of data while it is transferring data. This allows the system to perform powerful string operations at very high data rates. The transfer rates shown in Table 1 illustrate the improvement in system throughput that can be achieved with a DMA device.

INTEGRATION OF A Z80 DMA IN A Z8000 SYSTEM

A small, yet effective, Z8000 system can be built using currently available Z80 peripherals. The implementation of such a system is fully described in the Zilog application note A Small Z8000 System (document #03-8000-01). Previous discussion has proven the advantage of the addition of a DMA device to such a system. The rest of this article will describe the additional logic required to integrate the Z80 DMA into a Z8000-based system. By carefully selecting and implementing only those functions required, the designer can minimize the additional TTL logic. Since Z80 peripherals share common interface logic, it is not necessary to duplicate the logic when other Z80 peripherals are added to the system.

### Table 1. Maximum Data Transfer Rates

<table>
<thead>
<tr>
<th></th>
<th>Z80A CPU</th>
<th>Z80A DMA</th>
<th>Z8000 CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory to Memory</td>
<td>0.44 Mbytes/sec</td>
<td>0.44 Mbytes/sec</td>
<td>0.44 Mbytes/sec</td>
</tr>
<tr>
<td>1/O to 1/O</td>
<td>1.0 Mbytes/sec</td>
<td>1.0 Mbytes/sec</td>
<td>1.0 Mbytes/sec</td>
</tr>
<tr>
<td>1/O to Memory</td>
<td>2.0 Mbytes/sec</td>
<td>2.0 Mbytes/sec</td>
<td>2.0 Mbytes/sec</td>
</tr>
</tbody>
</table>

1 Continuous mode operation
* In Search/Transfer mode with external logic
** Requires external logic for word transfers

Figure 1 shows a block diagram of the interface requirements for a Z80 DMA device in a Z8000 system. The Small Z8000 System Application Note already implements part of the logic shown in Figure 1. These interface functions are common to other Z80 peripherals, such as the PIO, SIO and CTC. This includes the 3-state address buffers and bidirectional data buffers, which are used to demultiplex the system address and data buses. The DMA is connected to the demultiplexed address and data lines rather than being placed closer to the CPU. Other common functional blocks are the Status Decoder, I/O Decoder, and Z8000-to-Z80 Control Translator logic.
Since the Z80 DMA takes complete control of address and data buses during an operation, it generates Z80 CPU system-bus-compatible control signals. However, these signals are not compatible with the system bus control signals generated by Z8000 CPU, and a Z80-to-Z8000 Control Translator logic block is required to interface the DMA with the Z8000 system. In particular, the signals that need to be generated in order to effectively control the system bus are four status signals ST0-ST3, Byte/Word (B/W), Normal/System (N/S), Read/Write (R/W), Memory Request (MREQ), Data Strobe (DS), and Address Strobe (AS). The segmented Z8001 CPU generates a segment address and a 16-bit offset address within the segment. Since the DMA can only output 16 bits of address information, a Segment Register is required to store the segment information. The segment number is latched in this register by the Z8000 CPU prior to DMA operation. In memory-to-memory data transfers, the data to be moved must reside in the same 64K address space. However, in memory-to-I/O operations, when the block of data to be moved crosses a segment boundary, the operation requires the loading of a new segment number into the Segment Register before crossing the segment boundary. The Segment Register is shown in Figure 1.

A 4-bit Control Register that has been appropriately programmed by the Z8000 CPU before it enables the DMA is used to generate N/S, B/W, and W/DW signals. These three signals remain active throughout the DMA operation. The DMA provides two signals (MREQ and IORQ) that indicate whether a memory or an I/O address is being accessed. These signals are gated with signals generated by the Z8000 Status Decoder, which decodes the status signals ST0-ST3 to differentiate between memory and I/O accesses in the current CPU operation. Since the memory and I/O address spaces of the DMA are the same size, the MREQ and IORQ signals can be interchanged to generate other Z8000 control signals. The Write (WR) signal of the DMA is used to generate the R/W signal.

The timing relationship between the DMA control signals (IORQ, MREQ, RD, WR) and three of the Z8000 control signals (AS, DS, MREQ) is shown in Figure 2. In order to generate AS and DS from the DMA-generated control signals, the DMA must be operated in the variable cycle mode with a cycle length of four clock cycles. The DMA, however, can be allowed to run with an operational cycle of two clock cycles, if the memory controller can initiate and complete a memory transaction with the DMA's control signals instead of using AS and DS, and if the memory devices have the fast access times necessary for two-cycle transfers. Figure 3 illustrates the generation of AS, DS, and MREQ signals from DMA control signals RD, WR, and MREQ. The four clock cycle memory read or write operation of the DMA is translated to a three clock cycle CPU memory read or write operation with this logic. The DS signal is
When a dynamic RAM array needs to be refreshed, it becomes necessary to extend a DMA read or write cycle. This is achieved by activating the WAIT signal of the DMA. This signal is multiplexed with the Chip Enable (CE) signal in the device, since the DMA needs to be waited only when it is the bus master. The WAIT signal, however, is sampled only at fixed instances during a read or a write cycle and then only if the cycle is more than two clock cycles long when the programmable operational cycle feature is selected. Thus, in a three or four clock cycle Memory Read or Write, the WAIT line is sampled at the falling edge of the second clock, and on the falling edge of the third clock in a four clock cycle I/O Read or Write as illustrated in Figure 2. This implies that in order to be able to use the WAIT signal to extend the DMA operational cycle, the designer has to opt for four clock cycle transfers and use IORQ signal from the DMA to generate AS and DS signals, rather than the MREQ signal as shown in Figure 3. Since the memory and I/O spaces of the 280 DMA are 64K bytes each, the IORQ signal can be used to indicate a memory access and the MREQ signal to indicate I/O access.

Figure 2. Control Signal Timings

Figure 3. AS-.DS-.MREQ- Generation
The address translation logic, in conjunction with the data buffers, allows the DMA to perform byte, word or double word transfers. The designer has the option of selecting one or more of these data transfer modes. However, the hardware required to implement the functions increases as more options are selected. When only byte transfers are desired, no address translation logic or data buffering is needed, but, because the system data bus is 16-bits wide, an 8-bit bus transceiver buffer is required to enable the DMA to access the higher byte of the data bus (Figure 4). In this case, the DMA's address bus is directly connected to the system address bus. When 16-bit transfers are desired, the DMA address bus is shifted so that low address bit A0 is physically connected to system address bit SA1. In this case, A15 of the DMA is not used and SA0 is ignored by the memory controller. An 8-bit data buffer serves the purpose of storing the higher order data byte during the read cycle and driving it in the write cycle. This is illustrated in Figure 5. The 32-bit data transfer operation is similar to the 16-bit operation but requires two additional data buffers and the shifting of the address bus by an additional bit. These approaches, however, require that the same data bus width be used in data transfers between memory and an I/O device.

Figure 6 shows the address translation logic needed to do 8-, 16- and 32-bit data transfers. The CPU needs to set up two signals, B/W and W/DW, before enabling the DMA to determine the data transfer width. These two signals then control the shifting of the DMA's address bus for the generation of system addresses. Thus, while moving bytes, the two transparent latches are enabled and the DMA address bus remains unshifted. The data byte can be stored in any of the data buffers.
buffers (Figure 5) or by the DMA, depending upon the memory organization. To accomplish word or double word transfers, the address bus is shifted via the multiplexers by one or two bits, depending on the control signals. Only the four multiplexers and a data buffer are required to perform 8- and 16-bit data movements. Since the upper address bits from DMA are not used in 16- and 32-bit transfers, up to 32K words and 16K double words can be moved in a single DMA block transfer. To compensate for the shifting of these addresses, the actual port addresses are shifted right by one or two bits before being written to the DMA.

Figure 6. 8-, 16- or 32-Bit Data Transfer Address Translation Logic

USING THE SEARCH MODE

The search or search/transfer modes of the Z80 DMA need special interfacing consideration. Since the DMA can search for bytes only, the use of these functions is limited in a 16-bit environment without any support logic. Thus, when the DMA is set up to do 8-bit transfers, the hardware shown in Figure 4 allows searches on both halves of the data bus when the data bus is 16 bits wide. In the 16- and 32-bit transfer modes, however, the DMA can compare only the low-order data byte, and external hardware is required if any of the higher order data bytes need to be searched. When the hardware is set up to do 8-, 16- and 32-bit data transfers, the search mode can be used only if the memory control-interface logic. Also, the implementation of the extra logic needed to integrate the 8-bit DMA can be minimized by carefully selecting and implementing only necessary DMA functions that contribute to the improvement of overall system performance.

REFERENCES

INTRODUCTION

The Z8500 Family consists of universal peripherals that can interface to a variety of microprocessor systems that use a non-multiplexed address and data bus. Though similar to Z80 peripherals, the Z8500 peripherals differ in the way they respond to I/O and Interrupt Acknowledge cycles. In addition, the advanced features of the Z8500 peripherals enhance system performance and reduce processor overhead.

To design an effective interface, the user needs an understanding of how the Z80 Family interrupt structure works, and how the Z8500 peripherals interact with this structure. This application note provides basic information on the interrupt structures, as well as a discussion of the hardware and software considerations involved in interfacing the Z8500 peripherals to the Z80 CPUs. Discussions center around each of the following situations:

- Z80A 4 MHz CPU to Z8500 4 MHz peripherals
- Z80B 6 MHz CPU to Z8500A 6 MHz peripherals
- Z80H 8 MHz CPU to Z8500 4 MHz peripherals
- Z80H 8 MHz CPU to Z8500A 6 MHz peripherals

This application note assumes the reader has a strong working knowledge of the Z8500 peripherals; it is not intended as a tutorial.

CPU HARDWARE INTERFACING

The hardware interface consists of three basic groups of signals: data bus, system control, and interrupt control, described below. For more detailed signal information, refer to Zilog's Data Book, Universal Peripherals.

Data Bus Signals

D7-D0 Data Bus (bidirectional, 3-state). This bus transfers data between the CPU and the peripherals.

System Control Signals

A1-A0 Address Select Lines (optional). These lines select the port and/or control registers.

CE Chip Enable (input, active Low). CE is used to select the proper peripheral for programming. CE should be gated with TORE or MREQ to prevent spurious chip selects during other machine cycles.

RD* Read (input, active Low). RD activates the chip-read circuitry and gates data from the chip onto the data bus.

WR* Write (input, active Low). WR strobes data from the data bus into the peripheral.

*Chip reset occurs when RD and WR are active simultaneously.

Interrupt Control

INTACK Interrupt Acknowledge (input, active Low). This signal indicates an Interrupt Acknowledge cycle and is used with RD to gate the interrupt vector onto the data bus.

INT Interrupt Request (output, open-drain, active Low).
IEI  Interrupt Enable In (input, active High).

IEO  Interrupt Enable Out (output, active High).

These lines control the interrupt daisy chain for the peripheral interrupt response.

Z8500 I/O OPERATION

The Z8500 peripherals generate internal control signals from RD and WR. Since PCLK has no required phase relationship to RD or WR, the circuitry generating these signals provides time for metastable conditions to disappear.

The Z8500 peripherals are initialized for different operating modes by programming the internal registers. These internal registers are accessed during I/O Read and Write cycles, which are described below.

Read Cycle Timing

Figure 1 illustrates the Z8500 Read cycle timing. All register addresses and INTACK must remain stable throughout the cycle. If CE goes active after RD goes active, or if CE goes inactive before RD goes inactive, then the effective Read cycle is shortened.

Write Cycle Timing

Figure 2 illustrates the Z8500 Write cycle timing. All register addresses and INTACK must remain stable throughout the cycle. If CE goes active after WR goes active, or if CE goes inactive before WR goes inactive, then the effective Write cycle is shortened. Data must be available to the peripheral prior to the falling edge of WR.

PERIPHERAL INTERRUPT OPERATION

Understanding peripheral interrupt operation requires a basic knowledge of the Interrupt Pending (IP) and Interrupt Under Service (IUS) bits in relation to the daisy chain. Both Z80 and Z8500 peripherals are designed in such a way that no additional interrupts can be requested during an Interrupt Acknowledge cycle. This allows the interrupt daisy chain to settle, and ensures proper response of the interrupting device.

The IP bit is set in the peripheral when CPU intervention is required (such conditions as buffer empty, character available, error detection, or status changes). The Interrupt Acknowledge cycle does not necessarily reset the IP bit. This bit is cleared by a software command to the peripheral, or when the action that generated the interrupt is completed (i.e., reading a character, writing data, resetting errors, or changing the status). When the interrupt has been serviced, other interrupts can occur.

---

<table>
<thead>
<tr>
<th>ADDR</th>
<th>ADDRESS VALID</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTACK</td>
<td></td>
</tr>
<tr>
<td>CE</td>
<td></td>
</tr>
<tr>
<td>RD</td>
<td></td>
</tr>
<tr>
<td>DATA IN</td>
<td>DATA VALID</td>
</tr>
</tbody>
</table>

Figure 1. Z8500 Peripheral I/O Read Cycle Timing
The IUS bit indicates that an interrupt is currently being serviced by the CPU. The IUS bit is set during an Interrupt Acknowledge cycle if the IP bit is set and the IEI line is High. If the IEI line is Low, the IUS bit is not set, and the device is inhibited from placing its vector onto the data bus. In the Z80 peripherals, the IUS bit is normally cleared by decoding the RETI instruction, but can also be cleared by a software command (S10). In the Z8500 peripherals, the IUS bit is cleared only by software commands.

Z80 Interrupt Daisy-Chain Operation

In the Z80 peripherals, both the IP and IUS bits control the IEI line and the lower portion of the daisy chain.

When a peripheral's IP bit is set, its IEI line is forced Low. This is true regardless of the state of the IEI line. Additionally, if the peripheral's IUS bit is clear and its IEI line High, the NT line is also forced Low.

The Z80 peripherals sample for both MT and TORD active, and RD inactive to identify an Interrupt Acknowledge cycle. When MT goes active and RD is inactive, the peripheral detects an Interrupt Acknowledge cycle and allows its interrupt daisy chain to settle. When the TORD line goes active with MT active, the highest priority interrupting peripheral places its interrupt vector onto the data bus. The IUS bit is also set to indicate that the peripheral is currently under service. As long as the IUS bit is set, the IEI line is forced Low. This inhibits any lower priority devices from requesting an interrupt.

When the Z80 CPU executes the RETI instruction, the peripherals monitor the data bus and the highest priority device under service resets its IUS bit.

Z8500 Interrupt Daisy-Chain Operation

In the Z8500 peripherals, the IUS bit normally controls the state of the IEI line. The IP bit affects the daisy chain only during an Interrupt Acknowledge cycle. Since the IP bit is normally not part of the Z8500 peripheral interrupt daisy chain, there is no need to decode the RETI instruction. To allow for control over the daisy chain, Z8500 peripherals have a Disable Lower Chain (DLC) software command that pulls IEI Low. This can be used to selectively deactivate parts of the daisy chain regardless of the interrupt status. Table 1 shows the truth tables for the Z8500 interrupt daisy-chain control signals during certain cycles. Table 2 shows the interrupt state diagram for the Z8500 peripherals.

---

**Table 1. Z8500 Daisy-Chain Control Signals**

<table>
<thead>
<tr>
<th>Truth Table for Daisy Chain Signals During Idle State</th>
<th>Truth Table for Daisy Chain Signals During IACK Cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEI IP IUS IEO</td>
<td>IEI IP IUS IACK</td>
</tr>
<tr>
<td>0 X X X 0</td>
<td>0 X X X 0</td>
</tr>
<tr>
<td>1 X 0 1</td>
<td>1 1 X 0</td>
</tr>
<tr>
<td>1 X 1 0</td>
<td>1 X 1 0</td>
</tr>
<tr>
<td>1 0 0 1</td>
<td>1 0 0 1</td>
</tr>
</tbody>
</table>
Table 2. Z8500 Interrupt State Diagram

<table>
<thead>
<tr>
<th>Interrupt Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP Set</td>
</tr>
<tr>
<td>IEI High?</td>
</tr>
<tr>
<td>INT Active</td>
</tr>
<tr>
<td>&lt;------&gt; Wait for CPU INTACK Cycle</td>
</tr>
<tr>
<td>INTACK * IEI * RD</td>
</tr>
<tr>
<td>IUS Set</td>
</tr>
<tr>
<td>CPU Read, Write, or Reset IP</td>
</tr>
<tr>
<td>IP Cleared</td>
</tr>
<tr>
<td>IEO High?</td>
</tr>
<tr>
<td>IUS Cleared</td>
</tr>
</tbody>
</table>

Return to main program

The Z8500 peripherals use INTACK (Interrupt Acknowledge) for recognition of an Interrupt Acknowledge cycle. This pin, used in conjunction with RD, allows the Z8500 peripheral to gate its interrupt vector onto the data bus. An active RD signal during an Interrupt Acknowledge cycle performs two functions. First, it allows the highest priority device requesting an interrupt to place its interrupt vector on the data bus. Secondly, it sets the IUS bit in the highest priority device to indicate that the device is currently under service.

INPUT/OUTPUT CYCLES

Although Z8500 peripherals are designed to be as universal as possible, certain timing parameters differ from the standard Z80 timing. The following sections discuss the I/O interface for each of the Z80 CPUs and the Z8500 peripherals. Figure 5 depicts logic for the Z80A CPU to Z8500 peripherals (and Z80B CPU to Z8500A peripherals) I/O interface as well as the Interrupt Acknowledge interface. Figures 4 and 7 depict some of the logic used to interface the Z804 CPU to the Z8500 and Z8500A peripherals for the I/O and Interrupt Acknowledge interfaces. The logic required for adding additional Wait states into the timing flow is not discussed in the following sections.

Z80A CPU to Z8500 Peripherals

No additional Wait states are necessary during the I/O cycles, although additional Wait states can be inserted to compensate for timing delays that are inherent in a system. Although the Z80A timing parameters indicate a negative value for data valid prior to WR, this is a worse than "worst case" value. This parameter is based upon the longest (worst case) delay for data available from the falling edge of the CPU clock minus the shortest (best case) delay for CPU clock High to WR Low. The negative value resulting from these two parameters does not occur because the worst case of one parameter and the best case of the other do not occur within the same device. This indicates that the value for data available prior to WR will always be greater than zero.

All setup and pulse width times for the Z8500 peripherals are met by the standard Z80A timing. In determining the interface necessary, the CE signal to the Z8500 peripherals is assumed to be the decoded address qualified with the TORD signal.

Figure 3a shows the minimum Z80A CPU to Z8500 peripheral interface timing for I/O cycles. If additional Wait states are needed, the same number of Wait states can be inserted for both I/O Read and Write cycles to simplify interface logic. There are several ways to place the Z80A CPU into a Wait condition (such as counters or shift registers to count system clock pulses), depending upon whether or not the user wants to place Wait states in all I/O cycles, or only during Z8500 I/O cycles. Tables 3 and 4 list the Z8500 peripheral and the Z80A CPU timing parameters (respectively) of concern during the I/O cycles. Tables 5 and 6 list the equations used in determining if these parameters are satisfied. In generating these equations and the values obtained from them, the required number of Wait states was taken into account. The reference numbers in Tables 3 and 4 refer to the timing diagram in Figure 3a.
### Table 3. Z8500 Timing Parameters I/O Cycles

<table>
<thead>
<tr>
<th>Worst Case</th>
<th>Min</th>
<th>Max</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>6. TsA(WR)</td>
<td>80</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>1. TsA(RD)</td>
<td>80</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>2. TdA(DR)</td>
<td>590</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>TsCE1(WR)</td>
<td>0</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>TsCE1(RD)</td>
<td>0</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>4. TwRD1</td>
<td>390</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>8. TwWR1</td>
<td>390</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>3. TdRDf(DR)</td>
<td>255</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>7. TsDW(WR)</td>
<td>0</td>
<td></td>
<td>ns</td>
</tr>
</tbody>
</table>

### Table 4. Z80A Timing Parameters I/O Cycles

<table>
<thead>
<tr>
<th>Worst Case</th>
<th>Min</th>
<th>Max</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>TcC</td>
<td>250</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>TwCh</td>
<td>110</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>TfC</td>
<td>30</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>TdCr(A)</td>
<td>110</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>TdCr(RDf)</td>
<td>85</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>TdCr(IOQRf)</td>
<td>75</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>TdCr(WRF)</td>
<td>65</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>5. TsD(CF)</td>
<td>50</td>
<td></td>
<td>ns</td>
</tr>
</tbody>
</table>

### Table 5. Parameter Equations

<table>
<thead>
<tr>
<th>Z8500 Parameter</th>
<th>Z8500 Equation</th>
<th>Value</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>TsA(RD)</td>
<td>TcC-TdCr(A)</td>
<td>140</td>
<td>min ns</td>
</tr>
<tr>
<td>TdA(DR)</td>
<td>3TcC+TwCh-TdCr(A)-TsD(CF)</td>
<td>800</td>
<td>min ns</td>
</tr>
<tr>
<td>TdRDf(DR)</td>
<td>2TcC+TwCh-TsD(CF)</td>
<td>460</td>
<td>min ns</td>
</tr>
<tr>
<td>TwRD1</td>
<td>2TcC+TwCh+TfC-TdCr(RDF)</td>
<td>525</td>
<td>min ns</td>
</tr>
<tr>
<td>TsA(WR)</td>
<td>TcC-TdCr(A)</td>
<td>140</td>
<td>min ns</td>
</tr>
<tr>
<td>TsDW(WR)</td>
<td>&gt; 0 min</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TwWR1</td>
<td>2TcC+TwCh+TfC-TdCr(WRF)</td>
<td>560</td>
<td>min ns</td>
</tr>
</tbody>
</table>

### Table 6. Parameter Equations

<table>
<thead>
<tr>
<th>Z80A Parameter</th>
<th>Z80A Equation</th>
<th>Value</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>TsD(CF)</td>
<td>Address</td>
<td>160</td>
<td>min ns</td>
</tr>
<tr>
<td></td>
<td>3TcC+TwCh-TdCr(A)-TdA(DR)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2TcC+TwCh-TdCr(RDF)-TdRD(DR)</td>
<td>135</td>
<td>min ns</td>
</tr>
</tbody>
</table>
No additional Wait states are necessary during I/O cycles, although Wait states can be inserted to compensate for any system delays. Although the Z80B timing parameters indicate a negative value for data valid prior to WR, this is a worse than "worst case" value. This parameter is based upon the longest (worst case) delay for data available from the falling edge of the CPU clock minus the shortest (best case) delay for CPU clock High to WR Low. The negative value resulting from these two parameters does not occur because the worst case of one parameter and the best case of the other do not occur within the same device. This indicates that the value for data available prior to WR will always be greater than zero.

All setup and pulse width times for the Z8500A peripherals are met by the standard Z80B timing. In determining the interface necessary, the CE signal to the Z8500A peripherals is assumed to be the decoded address qualified with the TORD signal.

Figure 3a. Z80A CPU to Z8500 Peripheral Minimum I/O Cycle Timing
Figure 3b shows the minimum Z80B CPU to Z8500A peripheral interface timing for I/O cycles. If additional Wait states are needed, the same number of Wait states can be inserted for both I/O Read and I/O Write cycles in order to simplify interface logic. There are several ways to place the Z80B CPU into a Wait condition (such as counters or shift registers to count system clock pulses), depending upon whether or not the user wants to place Wait states in all I/O cycles, or only during Z8500A I/O cycles. Tables 7 and 8 list the Z8500A peripheral and the Z80B CPU timing parameters (respectively) of concern during the I/O cycles. Tables 9 and 10 list the equations used in determining if these parameters are satisfied. In generating these equations and the values obtained from them, the required number of Wait states was taken into account. The reference numbers in Tables 7 and 8 refer to the timing diagram of Figure 3b.

Figure 3b. Z80B CPU to Z8500A Peripheral Minimum I/O Cycle Timing
### Table 7. Z8500A Timing Parameters I/O Cycles

<table>
<thead>
<tr>
<th>Worst Case</th>
<th>Min</th>
<th>Max</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>6. TsA(WR) Address to WR Low Setup</td>
<td>80</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>1. TsA(RD) Address to RD Low Setup</td>
<td>80</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>2. TdA(DR) Address to Read Data Valid</td>
<td></td>
<td>420</td>
<td>ns</td>
</tr>
<tr>
<td>TsCE1(WR) CE Low to WR Low Setup</td>
<td>0</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>TsCE1(RD) CE Low to RD Low Setup</td>
<td>0</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>4. TwRD1 RD Low Width</td>
<td></td>
<td>250</td>
<td>ns</td>
</tr>
<tr>
<td>8. TwWR1 WR Low Width</td>
<td></td>
<td>250</td>
<td>ns</td>
</tr>
<tr>
<td>3. TdRDf(DR) RD Low to Read Data Valid</td>
<td></td>
<td>180</td>
<td>ns</td>
</tr>
<tr>
<td>7. TsDW(WR) Write Data to WR Low Setup</td>
<td></td>
<td>0</td>
<td>ns</td>
</tr>
</tbody>
</table>

### Table 8. Z80B Timing Parameters I/O Cycles

<table>
<thead>
<tr>
<th>Worst Case</th>
<th>Min</th>
<th>Max</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>TeC Clock Cycle Period</td>
<td>165</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>TwCh Clock Cycle High Width</td>
<td>65</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>TfC Clock Cycle Fall Time</td>
<td>20</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>TdCr(A) Clock High to Address Valid</td>
<td>90</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>TdCr(RDf) Clock High to RD Low</td>
<td>70</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>TdCr(IORQf) Clock High to IORQ Low</td>
<td>65</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>TdCr(WRF) Clock High to WR Low</td>
<td>60</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>5. TsD(Cf) Data to Clock Low Setup</td>
<td>40</td>
<td></td>
<td>ns</td>
</tr>
</tbody>
</table>

### Table 9. Parameter Equations

<table>
<thead>
<tr>
<th>Z8500A Parameter</th>
<th>Z80B Parameter</th>
<th>Placeholders</th>
<th>Value</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>TeA(RD)</td>
<td>TeC-TdCr(A)</td>
<td>&gt;75 min</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>TdA(DR)</td>
<td>3TcC+TwCh-TdCr(A)-TeD(Cf)</td>
<td>430 min</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>TdRDf(DR)</td>
<td>2TcC+TwCh-TsD(Cf)</td>
<td>345 min</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>TwRD1</td>
<td>2TcC+TwCh+TfC-TdCr(RDf)</td>
<td>325 min</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>TsA(WR)</td>
<td>TeC-TdCr(A)</td>
<td>75 min</td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>TsDW(WR)</td>
<td>&gt; 0 min</td>
<td></td>
<td>ns</td>
<td></td>
</tr>
<tr>
<td>TwWR1</td>
<td>2TcC+TwCh+TfC-TdCr(WRF)</td>
<td>352 min</td>
<td>ns</td>
<td></td>
</tr>
</tbody>
</table>

### Table 10. Parameter Equations

<table>
<thead>
<tr>
<th>Z80B Parameter</th>
<th>Z8500A Parameter</th>
<th>Placeholders</th>
<th>Value</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>TsD(Cf)</td>
<td>Address</td>
<td></td>
<td>50 min</td>
<td>ns</td>
</tr>
<tr>
<td></td>
<td>3TcC+TwCh-TdCr(A)-TdA(DR)</td>
<td></td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td></td>
<td>RD</td>
<td></td>
<td>2TcC+TwCh-TdCr(RDf)-TdRD(DR)</td>
<td>75 min</td>
</tr>
</tbody>
</table>
During an I/O Read cycle, there are three Z8500 parameters that must be satisfied. Depending upon the loading characteristics of the RD signal, the designer may need to delay the leading (falling) edge of RD to satisfy the Z8500 timing parameter TsA(RD) (Address Valid to RD Setup). Since Z80H timing parameters indicate that the RD signal may go Low after the falling edge of T2, it is recommended that the rising edge of the system clock be used to delay RD (if necessary). The CPU must also be placed into a Wait condition long enough to satisfy TdA(DR) (Address Valid to RD Low to Read Data Valid Delay).

During an I/O Write cycle, there are three other Z8500 parameters that must be satisfied. Depending upon the loading characteristics of the WR signal and the data bus, the designer may need to delay the leading (falling) edge of WR to satisfy the Z8500 timing parameters TsA(WR) (Address Valid to WR Setup) and TsDW(WR) (Data Valid Prior to WR setup). Since Z80H timing parameters indicate that the WR signal may go Low after the falling edge of T2, it is recommended that the rising edge of the system clock be used to delay WR (if necessary). This delay will ensure that both parameters are satisfied. The CPU must also be placed into a Wait condition long enough to satisfy TwWR1 (WR Low Pulse Width). Assuming that the WR signal is delayed, only two additional Wait states are needed during an I/O Write cycle when interfacing the Z80H CPU to the Z8500 peripherals.

To simplify the I/O interface, the designer can use the same number of Wait states for both I/O Read and I/O Write cycles. Figure 3c shows the minimum Z80H CPU to Z8500 peripheral interface timing for the I/O cycles (assuming that the same number of Wait states are used for both cycles and that both RD and WR need to be delayed). Figure 4 shows two circuits that can be used to delay the leading (falling) edge of either the RD or the WR signals. There are several ways to place the Z80A CPU into a Wait condition (such as counters or shift registers to count system clock pulses), depending upon whether or not the user wants to place Wait states in all I/O cycles, or only during Z8500 I/O cycles. Tables 4 and 11 list the Z8500 peripheral and the Z80H CPU timing parameters (respectively) of concern during the I/O cycles. Tables 14 and 15 list the equations used in determining if these parameters are satisfied. In generating these equations and the values obtained from them, the required number of Wait states was taken into account. The reference numbers in Tables 4 and 11 refer to the timing diagram of Figure 3c.

Table 11. Z80H Timing Parameter I/O Cycles

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Equation</th>
<th>Min</th>
<th>Max</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>TcC</td>
<td>Clock Cycle Period</td>
<td>125</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>TwCh</td>
<td>Clock Cycle High Width</td>
<td>55</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>fFC</td>
<td>Clock Cycle Fall Time</td>
<td>10</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>TdCr(A)</td>
<td>Clock High to Address Valid</td>
<td>80</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>TdCr(RDF)</td>
<td>Clock High to RD Low</td>
<td>60</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>TdCr(IORQ)</td>
<td>Clock High to IORQ Low</td>
<td>55</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>TdW(WR)</td>
<td>Clock High to WR Low</td>
<td>55</td>
<td></td>
<td>ns</td>
</tr>
<tr>
<td>5. TdD(Cf)</td>
<td>Data to Clock Low Setup</td>
<td>30</td>
<td></td>
<td>ns</td>
</tr>
</tbody>
</table>

Table 12. Parameter Equations

<table>
<thead>
<tr>
<th>Z8500 Parameter</th>
<th>Z80H Equation</th>
<th>Value</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>TsA(RD)</td>
<td>2TcC-TdCr(A)</td>
<td>170 m</td>
<td>ns</td>
</tr>
<tr>
<td>TdA(DR)</td>
<td>6TcC+TwCh-TdCr(A)-TsD(Cf)</td>
<td>695</td>
<td>min ns</td>
</tr>
<tr>
<td>TdRDF(DR)</td>
<td>4TcC+TwCh-TsD(Cf)</td>
<td>523</td>
<td>min ns</td>
</tr>
<tr>
<td>TwRD1</td>
<td>4TcC+TwCh+fFC-TdCr(RDF)</td>
<td>503</td>
<td>min ns</td>
</tr>
<tr>
<td>TsA(WR)</td>
<td>WR - delayed</td>
<td>2TcC-TdCr(A)</td>
<td>170 min ns</td>
</tr>
<tr>
<td>TsDW(WR)</td>
<td></td>
<td>&gt; 0 min</td>
<td>ns</td>
</tr>
<tr>
<td>TwWR1</td>
<td>4TcC+TwCh+fFC</td>
<td>563</td>
<td>min ns</td>
</tr>
</tbody>
</table>

2-37
Figure 3c. Z800 CPU to Z8500 Peripheral Minimum I/O Cycle Timing
Z80H CPU to Z8500A Peripherals

During an I/O Read cycle, there are three Z8500A parameters that must be satisfied. Depending upon the loading characteristics of the RD signal, the designer may need to delay the leading (falling) edge of RD to satisfy the Z8500A timing parameter TsA(RD) (Address Valid to RD Setup). Since Z80H timing parameters indicate that the RD signal may go low after the falling edge of $\tau_2$, it is recommended that the rising edge of the system clock be used to delay RD (if necessary). The CPU must also be placed into a Wait condition long enough to satisfy TdA(DR) (Address Valid to Read Data Valid Delay) and TdRDf(DR) (RD Low to Read Data Valid Delay). Assuming that the RD signal is delayed, then only one additional Wait state is needed during an I/O Read cycle when interfacing the Z80H CPU to the Z8500A peripherals.

During an I/O Write cycle, there are three other Z8500A parameters that have to be satisfied. Depending upon the loading characteristics of the WR signal and the data bus, the designer may need to delay the leading (falling) edge of WR to satisfy the Z8500A timing parameters TsA(WR) (Address Valid to WR Setup) and TsDW(WR) (Data Valid Prior to WR Setup). Since Z80H timing parameters indicate that the WR signal may go low after the falling edge of $\tau_2$, it is recommended that the rising edge of the system clock be used to delay WR (if necessary). This delay will ensure that both parameters are satisfied. The CPU must also be placed into a Wait condition long enough to satisfy TwWRl (WR Low Pulse Width). Assuming that the WR signal is delayed, then only one additional Wait state is needed during an I/O Write cycle when interfacing the Z80H CPU to the Z8500A peripherals.

Figure 3d shows the minimum Z80H CPU to Z8500A peripheral interface timing for the I/O cycles (assuming that the same number of Wait states are used for both cycles and that both RD and WR need to be delayed). Figure 4 shows two circuits that may be used to delay the leading (falling) edge of either the RD or the WR signals. There are several methods used to place the Z80A CPU into a Wait condition (such as counters or shift registers to count system clock pulses), depending upon whether or not the user wants to place Wait states in all I/O cycles, or only during Z8500A I/O cycles. Tables 7 and 11 list the Z8500A peripheral and the Z80H CPU timing parameters (respectively) of concern during the I/O cycles. Tables 14 and 15 list the equations used in determining if these parameters are satisfied. In generating these equations and the values obtained from them, the required number of Wait states was taken into account. The reference numbers in Tables 4 and 11 refer to the timing diagram of Figure 3d.

Table 13. Parameter Equations

<table>
<thead>
<tr>
<th>Z80H Parameter</th>
<th>Z8500 Equation</th>
<th>Value</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>TsD(Cf)</td>
<td>Address</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6TcC+TwCh-TdCr(A)-TdA(DR)</td>
<td>300 min</td>
<td>ns</td>
</tr>
</tbody>
</table>

Table 14. Parameter Equations

<table>
<thead>
<tr>
<th>Z8500A Parameter</th>
<th>Z80H Equation</th>
<th>Value</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>TsA(RD)</td>
<td>2TcC-TdCr(A)</td>
<td>170 min</td>
<td>ns</td>
</tr>
<tr>
<td>TdA(DR)</td>
<td>6TcC+TwCh-TdCr(A)-TsD(Cf)</td>
<td>695 min</td>
<td>ns</td>
</tr>
<tr>
<td>TdRDf(DR)</td>
<td>4TcC+TwCh-TsD(Cf)</td>
<td>525 min</td>
<td>ns</td>
</tr>
<tr>
<td>TwRDl</td>
<td>4TcC+TwCh+TFC-TdCr(RDF)</td>
<td>503 min</td>
<td>ns</td>
</tr>
<tr>
<td>TsA(WR)</td>
<td>WR - delayed</td>
<td>170 min</td>
<td>ns</td>
</tr>
<tr>
<td>TwWRl</td>
<td>2TcC-TdCr(A)</td>
<td>&gt; 0 min</td>
<td>ns</td>
</tr>
<tr>
<td>TsDW(WR)</td>
<td>2TcC+TwCh+TFC</td>
<td>313 min</td>
<td>ns</td>
</tr>
</tbody>
</table>
Figure 3d. Z80H CPU to Z8500A Peripheral Minimum I/O Cycle Timing
Figure 4. Delaying RD or WR

Table 15. Parameter Equations

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Z80H Equation</th>
<th>Z8500A Equation</th>
<th>Value</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>TeD(Cf)</td>
<td>Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4TcC + TwCh - TdCr(A) - TdA(DR)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RD - delayed</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2TcC + TwCh - TdRD(DR)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>55 min</td>
<td>ns</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>125 min</td>
<td>ns</td>
</tr>
</tbody>
</table>
INTERRUPT ACKNOWLEDGE CYCLES

The primary timing differences between the Z80 CPUs and Z8500 peripherals occur in the Interrupt Acknowledge cycle. The Z8500 timing parameters that are significant during Interrupt Acknowledge cycles are listed in Table 16, while the Z80 parameters are listed in Table 17. The reference numbers in Tables 16 and 17 refer to Figures 6, 8a, and 8b.

If the CPU and the peripherals are running at different speeds (as with the Z80H interface), the INTACK signal must be synchronized to the peripheral clock. Synchronization is discussed in detail under Interrupt Acknowledge for Z80H CPU to Z8500/B500A Peripherals.

During an Interrupt Acknowledge cycle, Z8500 peripherals require both INTACK and RD to be active at certain times. Since the Z80 CPUs do not issue either INTACK or RD, external logic must generate these signals.

Generating these two signals is easily accomplished, but the Z80 CPU must be placed into a Wait condition until the peripheral interrupt vector is valid. If more peripherals are added to the daisy chain, additional Wait states may be necessary to give the daisy chain time to settle. Sufficient time between INTACK active and RD active should be allowed for the entire daisy chain to settle.

Since the Z8500 peripheral daisy chain does not use the IP flag except during interrupt acknowledge, there is no need for decoding the RETI instruction used by the Z80 peripherals. In each of the Z8500 peripherals, there are commands that reset the individual IUS flags.

EXTERNAL INTERFACE LOGIC

The following sections discuss external interface logic required during Interrupt Acknowledge cycles for each interface type.

CPU/Peripheral Same Speed

Figure 5 shows the logic used to interface the Z80A CPU to the Z8500 peripherals and the Z80B CPU to Z8500A peripherals during an Interrupt Acknowledge cycle. The primary component in this logic is the Shift register (74LS164), which generates INTACK, READ, and WAIT.

<table>
<thead>
<tr>
<th>Table 16. Z8500 Timing Parameters Interrupt Acknowledge Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Worst Case</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>1. TAIA(PC)</td>
</tr>
<tr>
<td>ThIA(PC)</td>
</tr>
<tr>
<td>2. TDIAi(RD)</td>
</tr>
<tr>
<td>TwRDA</td>
</tr>
<tr>
<td>3. TDRA(RDA)</td>
</tr>
<tr>
<td>TAEI(RDA)</td>
</tr>
<tr>
<td>THei(RDA)</td>
</tr>
<tr>
<td>TDIEI(IE)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 17. Z80 CPU Timing Parameters Interrupt Acknowledge Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Worst Case</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>1. TdC(M1f)</td>
</tr>
<tr>
<td>TdM1f(IORQF)</td>
</tr>
<tr>
<td>4. Tsd(Cr)</td>
</tr>
<tr>
<td>*</td>
</tr>
<tr>
<td>Z80A:</td>
</tr>
<tr>
<td>Z80B:</td>
</tr>
<tr>
<td>Z80H:</td>
</tr>
</tbody>
</table>

2-42
During I/O and normal memory access cycles, the Shift register remains cleared because the WR signal is inactive. During opcode fetch cycles, also, the Shift register remains cleared, because only Os can be clocked through the register. Since Shift register outputs are Low, READ, WRITE, and WAIT are controlled by other system logic and gated through the AND gates (74LS11). During I/O and normal memory access cycles, READ and WRITE are active as a result of the system RD and WR signals (respectively) becoming active. If system logic requires that the CPU be placed into a Wait condition, the WAIT' signal controls the CPU. Should it be necessary to reset the system, RESET causes the interface logic to generate both READ and WRITE (the Z8500 peripheral Reset condition).

Normally an Interrupt Acknowledge cycle is indicated by the Z80 CPU when WR and TORQ are both active (which can be detected on the third rising clock edge after T1). To obtain an early indication of an Interrupt Acknowledge cycle, the Shift register decodes an active WR in the presence of an inactive MREQ on the rising edge of T2.

During an Interrupt Acknowledge cycle, the INTACK signal is generated on the rising edge of T2. Since it is the presence of INTACK and an active READ that gates the interrupt vector onto the data bus, the logic must also generate READ at the proper time. The timing parameter of concern here is Td1Ai(RD) [INTACK to RD (Acknowledge) Low Delay]. This time delay allows the interrupt daisy chain to settle so that the device requesting the interrupt can place its interrupt vector onto the data bus. The Shift register allows a sufficient time delay from the generation of INTACK before it generates READ. During this delay, it places the CPU into a Wait state until the valid interrupt vector can be placed onto the data bus. If the time between these two signals is insufficient for daisy chain settling, more time can be added by taking READ and WAIT from a later position on the Shift register.

Figure 6 illustrates Interrupt Acknowledge cycle timing resulting from the Z80A CPU to Z8500 peripheral and the Z80B CPU to Z8500A peripheral interface. This timing comes from the logic illustrated in Figure 5, which can be used for both interfaces. Should more Wait states be required, the additional time can be calculated in terms of system clocks, since the CPU clock and PCLK are the same.
Figure 6. Z80A/Z80B CPU to Z8500/Z8500A Peripheral Interrupt Acknowledge Interface Timing

Z80H CPU to Z8500/Z8500A Peripherals

Figure 7 depicts logic that can be used in interfacing the Z80H CPU to the Z8500/Z8500A peripherals. This logic is the same as that shown in Figure 5, except that a synchronizing flip-flop is used to recognize an Interrupt Acknowledge cycle. Since Z8500 peripherals do not rely upon PCLK except during Interrupt Acknowledge cycles, synchronization need occur only at that time. Since the CPU and the peripherals are running at different speeds, INTACK and RD must be synchronized to the Z8500 peripherals clock.

During I/O and normal memory access cycles, the synchronizing flip-flop and the Shift register remain cleared because the \( \overline{MT} \) signal is inactive. During opcode fetch cycles, the flip-flop and the Shift register again remain cleared, but this time because the \( MREQ \) signal is active. The synchronizing flip-flop allows an Interrupt Acknowledge cycle to be recognized on the rising edge of \( T_2 \) when \( MT \) is active and \( MREQ \) is inactive, generating the INTA signal. When INTA is active, the Shift register can clock and generate INTACK to the peripheral and WAIT to the CPU. The Shift register delays the generation of READ to the peripheral until the daisy chain settles. The WAIT signal is removed when sufficient time has been allowed for the interrupt vector data to be valid.

Figure 8a illustrates Interrupt Acknowledge cycle timing for the Z80H CPU to Z8500 peripheral interface. Figure 8b illustrates Interrupt Acknowledge cycle timing for the Z80H CPU to Z8500A peripheral interface. These timings result from the logic in Figure 7. Should more Wait states be required, the needed time should be calculated in terms of PCLKs, not CPU clocks.

Z80 CPU to Z80 and Z8500 Peripherals

In a Z80 system, a combination of Z80 peripherals and Z8500 peripherals can be used compatibly. While there is no restriction on the placement of the Z8500 peripherals in the daisy chain, it is recommended that they be placed early in the chain to minimize propagation delays during RETI cycles.

During an Interrupt Acknowledge cycle, the IEO line from the Z8500 peripherals changes to reflect the interrupt status. Time should be allowed for this change to ripple through the remainder of the daisy chain before activating IORQ to the Z80 peripherals, or READ to the Z8500 peripherals.
During the RETI cycles, the IEO line from the Z8500 peripherals does not change state as in the Z80 peripherals. As long as the peripherals are at the top of the daisy chain, propagation delays are minimized.

The logic necessary to create the control signals for both Z80 and Z8500 peripherals is shown in Figure 7. This logic delays the generation of \( \text{IORQ} \) to the Z80 peripherals by the same amount of time necessary to generate \( \text{READ} \) for the Z8500 peripherals. Timing for this logic during an Interrupt Acknowledge cycle is depicted in Figure 10.
Figure 8a. Z80H CPU to Z8500 Peripheral Interrupt Acknowledge Interface Timing
Figure 8b. Z80H CPU to Z8500A Peripheral Interrupt Acknowledge Interface Timing
Figure 9. Z80 and Z8500 Peripheral Interrupt Acknowledge Interface Logic
Figure 10. 280 and 28500 Peripheral Interrupt Acknowledge Interface Timing
SOFTWARE CONSIDERATIONS -- POLLED OPERATION

There are several options available for servicing interrupts on the Z8500 peripherals. Since the vector or IP registers can be read at any time, software can be used to emulate the Z80 interrupt response. The interrupt vector read reflects the interrupt status condition even if the device is programmed to return a vector that does not reflect the status change (SAV or VIS is not set). The code below is a simple software routine that emulates the Z80 vector response operation.

**Z80 Vector Interrupt Response, Emulation by Software**

;This code emulates the Z80 vector interrupt operation by reading the device interrupt vector and forming an address from a vector table. It then executes an indirect jump to the interrupt service routine.

<table>
<thead>
<tr>
<th>INDX:</th>
<th>LD A,CIVREG ;CURRENT INT. VECT. REG.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>OUT (CTRL),A ;WRITE REG. PTR.</td>
</tr>
<tr>
<td></td>
<td>IN A,(CTRL) ;READ VECT. REG.</td>
</tr>
<tr>
<td></td>
<td>INC A ;VALID VECTOR?</td>
</tr>
<tr>
<td></td>
<td>RET Z ;NO INT - RETURN</td>
</tr>
<tr>
<td></td>
<td>AND 00001110B ;MASK OTHER BITS</td>
</tr>
<tr>
<td></td>
<td>LD E,A</td>
</tr>
<tr>
<td></td>
<td>LD D,0 ;FORM INDEX VALUE</td>
</tr>
<tr>
<td></td>
<td>LD HL,VECTAB</td>
</tr>
<tr>
<td></td>
<td>ADD HL,DE ;ADD VECT. TABLE ADDR.</td>
</tr>
<tr>
<td></td>
<td>LD A,(HL) ;GET LOW BYTE</td>
</tr>
<tr>
<td></td>
<td>INC HL</td>
</tr>
<tr>
<td></td>
<td>LD H,(HL) ;GET HIGH BYTE</td>
</tr>
<tr>
<td></td>
<td>LD L,A ;FORM ROUTINE ADDR.</td>
</tr>
<tr>
<td></td>
<td>JP (HL) ;JUMP TO IT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VECTAB:</th>
<th>DEFW INT1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DEFW INT2</td>
</tr>
<tr>
<td></td>
<td>DEFW INT3</td>
</tr>
<tr>
<td></td>
<td>DEFW INT4</td>
</tr>
<tr>
<td></td>
<td>DEFW INT5</td>
</tr>
<tr>
<td></td>
<td>DEFW INT6</td>
</tr>
<tr>
<td></td>
<td>DEFW INT7</td>
</tr>
<tr>
<td></td>
<td>DEFW INT8</td>
</tr>
</tbody>
</table>
A SIMPLE Z80-Z8500 SYSTEM

The Z8500 devices interface easily to the Z80 CPU, thus providing a system of considerable flexibility. Figure 11 illustrates a simple system using the Z80A CPU and the Z8536 Counter/Timer and Parallel I/O Unit (CIO) in a mode 1 or non-interrupt environment. Since interrupt vectors are not used, the INTACK line is tied High and no additional logic is needed. Because the CIO can be used in a polled interrupt environment, the INT pin is connected to the CPU. The Z80 should not be set for mode 2 interrupts since the CIO will never place a vector onto the data bus. Instead, the CPU should be placed into mode 1 interrupt mode and a global interrupt service routine can poll the CIO to determine what caused the interrupt to occur. In this system, the software emulation procedure described above is effective.

Figure 11. Z80 to Z8500 Simple System Mode 1 Interrupt or Non-Interrupt Structure

Additional Information - Zilog Publications

1. Z80 CPU Technical Manual (03-0029-01)  
2. Z80 DMA Technical Manual (00-2013-A0)  
3. Z80 PIO Technical Manual (03-0008-01)  
4. Z80 CTC Technical Manual (03-0036-02)  
5. Z80 SIO Technical Manual (03-3033-01)  
6. Z80H CPU AC Characteristics (00-2293-01)  
7. Z80 Family Interrupt Structure Tutorial (611-1809-0003)  
8. Z8530 SCC Technical Manual (00-2057-01)  
9. Z8536 CIO Technical Manual (00-2091-01)  
10. Z8038 F10 Technical Manual (00-2051-01)  
11. Zilog 1982/83 Data Book (00-2034-02)
INTRODUCTION

As operating systems grow more sophisticated, application programs more complex, and the use of high-level languages even more prevalent, the need for increased memory addressing space and some form of memory protection becomes critical.

The memory space requirements of many microprocessor applications have grown beyond the 64K byte addressing range of today's 8-bit microprocessors. While the available 16-bit processors offer dramatically increased memory addressing capabilities, the conversion to these products often cannot be justified. For example, in many cases an application might be better suited for 8-bit processing, and switching to a 16-bit processor could result in a costlier and less efficient implementation. Perhaps even more serious is the problem of software incompatibility that occurs when changing microprocessors. An ideal solution is one that both extends memory addressing space and is object code compatible with the user's existing software.

An additional requirement placed on the user by today's increasingly complex software is that of maintaining system integrity. In order to ensure this integrity, various parts of the system software must be protected from illegal access. Although memory protection features are an important part of memory management, they are not found on most microprocessors.

MEMORY MANAGEMENT TECHNIQUES

Before discussing the techniques used to expand the addressing space and provide memory protection, the concept of logical and physical addresses and of pages in memory needs to be explained. The logical address is the address generated by the microprocessor, and the physical address is the address received by the system memory. In a microprocessor system with no memory management, the physical address is the same as the logical address (Figure 1, section a). In a microprocessor system with memory management, the logical address generated by the processor is translated, or expanded, by the Memory Management Unit (MMU) before being sent to the system memory as the physical address (Figure 1, section b). For example, the 16-bit logical address of the Z80 could easily be expanded by an MMU to a 24-bit address.

This application note describes a way in which the Z80 user can increase memory addressing space to 16M and incorporate memory protection features while maintaining object code compatibility with application software. The memory management techniques employed here are a subset of those used by the Z800 series of microprocessors soon to be released by Zilog. These techniques provide a direct path to the implementation of some Z800 features before the fully-integrated solution is available.
While there are many techniques that can be used to implement the address translation process, this application note considers the paging technique only. Two concepts are essential to the comprehension of paging: that of a logical page, which is a section of the address space of the microprocessor; and that of a page frame, which is a section of physical memory. A page frame is simply a fixed-length block of physical memory. For the purposes of this application note, a page frame consists of a 4K (4096 bytes) block of physical memory. Each byte of a page frame can be uniquely addressed by a combination of 12 address lines (12 bits specify 4096 bytes). The 64K logical address space of an 8-bit microprocessor contains 16 logical pages, and a 16M physical address space contains 4096 (4K) page frames. A memory management system maps the 16 logical pages that the microprocessor "sees" into 16 of the 4K page frames in the 16M physical memory (Figure 2). By partitioning the physical memory space into 4K page frames, both memory address space expansion and memory protection can be easily accomplished.
Memory address space expansion consists of taking a 16-bit logical address output by the microprocessor and generating from that a 24-bit physical address. The logical address is divided into two parts, a 12-bit displacement field and a 4-bit index field. The index field is used to select one of 16 registers known as page descriptor registers. Each page descriptor register contains 12 bits of addressing information, which is used to identify a page frame in physical memory. The page descriptor registers reside in the I/O space of the system and are maintained by the operating system. The physical address is generated by concatenating the 12 bits of page descriptor information from the selected page descriptor register with the 12-bit displacement field of the logical address. Therefore, when the microprocessor places a 16-bit logical address on the Address bus, the lower 12 bits (A0-A11) of the address are presented to the physical memory and Address bits A12-A15 are used to select one of the 16 page descriptor registers. The 12 bits of address contained in the selected register are placed on the bus to form the upper 12 bits of the physical Address (A12-A23). This process is shown in Figure 3.

The 16 page descriptor registers allow the user to access 16 separate page frames (64K bytes of active memory) at any one time. If it becomes necessary to access a page frame other than one of the 16 that are currently active, the operating system simply uses an I/O instruction to load a new page frame value into the appropriate page descriptor register. If the page descriptor registers are loaded with hex 000-00F, the resultant addressing is exactly the same as if the address space expansion were not present (i.e., the 24-bit physical Address bus addresses memory locations hex 000000-00FFFF).

**Memory Protection**

The memory protection features are implemented by using attributes associated with each page frame of memory. This is accomplished by assigning four bits of attributes to each page descriptor register. The page descriptor registers are 16 (rather than 12) bits wide. When a page descriptor register is selected by Address bits A12-A15, both the address and attribute information corresponding to that particular page frame is accessed. Attribute bits are used by external circuitry in the memory management system to monitor the types of accesses made to the page frames and to record information about the use of the page blocks. The attribute bits are the Valid bit, Write-Protect bit, and Modified bit, with one bit reserved for future use. A complete page descriptor register is shown in Figure 4.

The Valid bit is used to indicate if the page frame of memory associated with that particular page descriptor register can be accessed. This bit can be read from or written to by performing an I/O read or write to the appropriate page descriptor register. If the Valid bit of a page register is set to 1, it can be used to access memory. If the bit is cleared to 0, a memory access to that register is invalid. When an invalid access is made, an interrupt is generated and the address that caused the invalid access is saved for processing by the interrupt service routine.

The Write-Protect bit is used to assign read-only attributes to page frames of memory. Like the Valid bit, the Write-Protect bit can be read from or written to by the user. If the bit is set to 1, the memory is write-protected and an interrupt occurs if a write to memory is attempted. When the Write-Protect bit is cleared to 0, both read and write operations can be performed. This bit
is useful in a system in which multiple processors share common memory, or in which an operating system needs to be protected from accidental writes by an executing program.

The Modified bit is a status bit that is automatically set whenever a write is performed to a logical address within the page frame. It can be cleared only by reloading a 0 into the appropriate lower bit of the page descriptor register. The Modified bit is used to indicate if the page frame has been used for a memory access and is helpful in determining whether the information in the page frame needs to be copied to secondary storage before using the page frame for another purpose.

LOADING PAGE DESCRIPTOR REGISTERS

The page descriptor registers reside in the microprocessor's I/O space and are accessed by the microprocessor's I/O instructions. Each register is 16 bits long and so must be read to or written from twice in order to access the full register. To facilitate this double access, two I/O addresses are assigned to each page descriptor register: one for the upper byte and one for the lower byte. The assigned I/O addresses are listed in Table 1. The page descriptor registers can be accessed either individually or (by using the microprocessor's Block I/O instructions) as a block in I/O space.

Due to the uncertain state of the register content at power-up, certain provisions are necessary to ensure that the system behaves in a predictable manner. A bypass mechanism known as Pass mode enables the microprocessor to begin its initialization as if no memory management circuitry were present. In Pass mode, logical Address bits A12-A15 are passed on to physical Address bits A12-A15 and the physical Address bits A16-A23 are set Low. After initializing the page descriptor registers, the microprocessor can then enter Address Translation mode.

Table 1. I/O Port Registers

<table>
<thead>
<tr>
<th>Port Address</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>X X 0 0</td>
<td>System control port</td>
</tr>
<tr>
<td>X X 0 3</td>
<td>Page fault and system status</td>
</tr>
<tr>
<td>X X 1 0</td>
<td>Page descriptor register 0 (low byte)</td>
</tr>
<tr>
<td>X X 1 1</td>
<td>Page descriptor register 0 (high byte)</td>
</tr>
<tr>
<td>X X 1 2</td>
<td>Page descriptor register 1 (low byte)</td>
</tr>
<tr>
<td>X X 1 3</td>
<td>Page descriptor register 1 (high byte)</td>
</tr>
<tr>
<td>X X 1 4</td>
<td>Page descriptor register 2 (low byte)</td>
</tr>
<tr>
<td>X X 1 5</td>
<td>Page descriptor register 2 (high byte)</td>
</tr>
<tr>
<td>X X 2 E</td>
<td>Page descriptor register 15 (low byte)</td>
</tr>
<tr>
<td>X X 2 F</td>
<td>Page descriptor register 15 (high byte)</td>
</tr>
</tbody>
</table>
IMPLEMENTATION OF MEMORY MANAGEMENT TECHNIQUES

Implementation of the memory management techniques described above for the Z80 consists of circuitry for the memory address space expansion and memory protection features, as well as the necessary logic for power-up and interrupt-handling.

The memory address space expansion circuitry is based on the 74S612 Memory Mapper. This TTL circuit contains sixteen 12-bit registers which are used as page descriptor registers. Because the Memory Mapper’s registers are only 12 bits wide, sixteen 4-bit registers must be added to utilize the protection features. These 4-bit registers are added in the form of a 16 x 4 RAM (74S219) and an associated multiplexer (74S257). The registers contained in the RAM form the basis on which the attribute bits are associated with each page frame. These registers and the mapper registers are loaded at the same time, and together they form a set of 16-bit registers.

A functional block diagram of the circuit is shown in Figure 5. The diagram shows two address paths to the register set through the multiplexer. Input pins RS0-RS3 select a register for reading or loading during an I/O operation, and pins MA0-MA3 are used to generate a physical address. Logical address bits A12-A15 from the microprocessor are the input signals to the map address inputs MA0-MA3.

Figure 5. Memory Manager Block Diagram
The 74S612 Memory Mapper's Pass mode of operation is slightly different from the Pass mode previously described, and provisions must be made for it to operate in the required manner. In Pass mode, the 74S612 places the upper four bits of the logical address (A12-A15) on what corresponds to bits A20-A23 of the physical address while holding bits A12-A19 Low. This results in a physical address that is different from the logical address and makes Pass mode not useable for initialization. To correct this problem, the registers are loaded with data that has been rearranged so that Pass mode operates properly for initialization, but remains transparent to the user. This is accomplished by arranging the data lines and address output lines as shown in Figures 6a and 6b.

Memory protection features are incorporated by examining the attribute bits in the page descriptor register associated with the page frame of memory being accessed. Writing to or reading from a block of memory whose Valid bit is cleared to 0 or attempting to write to a page of memory whose Write-Protect bit is set to 1 causes a fault and interrupts the CPU. The Valid bit is tested during every Read or Write cycle to ensure that operations on that block of memory can be performed. If a fault occurs, a nonmaskable interrupt is generated to the CPU and Address bits A12-A15 of the logical address are latched. If the page is valid and a write is requested, the Write-Protect bit is checked to see if the page of memory is write-protected. As in the case of an invalid access attempt (valid = 0), a write-protect fault causes a nonmaskable interrupt to be generated to the CPU, and logical Address bits A12-A15 are latched. Since in both cases logical bits A12-A15 are latched, the interrupt service routine can read these bits to determine which page descriptor register contains the attribute bits that caused the faults. Reading I/O port 03H causes the four Address bits to be placed on data lines D0-D3.

The memory management circuit has two modes of operation: Pass mode and Address Translation mode. When powered up, the circuit is in Pass mode and the system appears as an unmodified Z80. During Pass mode and Interrupt Acknowledge cycles, the nonmaskable interrupt is inhibited to prevent any undesired interrupts from occurring. Memory translation is enabled by writing a 00H to I/O port 00H, and Pass mode can be reestablished by writing a 01H to the same I/O port. The System mode can be determined by reading bit 4 of I/O port 03H.

The circuit shown in Figures 6a and 6b was tested by using a Zilog ZDS 1/40 Development System with ZAP (Zilog Analyzer Program). Since the ZDS 1/40 does not have I/O mapping capability, a user clock was built to provide a complete testing of I/O ports used in the system. Some useful subroutines that can be used by the memory management circuit are given in the appendix.

CONCLUSION

The scheme described provides memory expansion and memory protection by using a flexible paging mechanism. The scheme is compatible with both Z80 object code and the forthcoming Z800 design. It therefore bridges the capabilities of the two compatible microprocessor families and saves both circuit design and software conversion effort.
Figure 6a. Memory Expansion Hardware Schematic
Figure 6b. Memory Expansion Hardware Schematic (Continued)
Appendix A. Some Useful Subroutines

; ***********************************************************************
; **  RETURN FROM LOAD & JUMP **
; ** SUBROUTINE **
; ***********************************************************************

; THIS ROUTINE Prepares the return for the original call.
; It will put back the value of the page descriptor reg.
; which was used to access another 4K page. First it pops
; the return address of the one which called it. Next it
; pops the original return address into de then executes
; the jpinIt subroutine to jump back.
; Passed parameter:
;   iy => previous register data
;   ix => previous register address

callout:
  pop de ; throw the call away
  pop de ; orig. return address
  jp jpinIt

; ***********************************************************************
; **  LOAD THEN JUMP ROUTINE **
; ***********************************************************************

; This will load the register with predefined address
; then jump to that location by changing the content of
; stack pointer before return. The format is followed:

  __________^______________  __________^_________
  |         |         |          |         |         |
  |         |         |          |         |         |
  |--------- Attribute |
  |-------- A23-A12 |

  __________^______________  __________^_________
  |         |         |          |         |         |
  |         |         |          |         |         |
  |--------- -------------- |
  |-------------- A11-A0 |
  |--------------------- logical page (0-F) |

Passed param.:
  A23-A16 => h
  A15-A12 + 4 bits attribute => l
  logical page + a11-a8 => d
  a7-a0 => e
  ix => register addr. table
  iy => register data
Appendix A. Some Useful Subroutines (Continued)

; RETURN PARAM.:  
; PC=DE  
; IX => REGISTER ADDR. TABLE  
; IY => REGISTER SAVED DATA

JPINIT: CALL FINDRG  
CALL SWAP  
PUSH DE  
RET ; JUMP

FINDRG: LD C,D ; MOVE LOGICAL PAGE  
SRL C ; TO LOWER NIBBLE  
SRL C  
SRL C  
SRL C  
LD B,O  
ADD IX,BC ; IX POINTS TO THE  
RET ; REGISTER ADDRESS

; THIS ROUTINE ONLY SWAPS THE CONTENT OF 1 REGISTER

SWAP: LD C,(IX+O) ; C HAS THE ADDRESS  
LD L,(IY+O) ; NEW LOW BYTE  
LD H,(IY+1) ; NEW HI-BYTE  
IN B,(C)  
LD (IY+O),B ; SAVE LOW BYTE  
OUT (C),L ; WRITE LOW BYTE  
INC C  
IN B,(C)  
LD (IY+1),B ; SAVE HI-BYTE  
OUT (C),H ; WRITE HI-BYTE  
RET

; *******************************************************
; *** LOAD PAGE REGISTERS ***
; *** SUBROUTINE ***
; *******************************************************

; PASSED & RETURN PARAMETERS:  
; POINTER TO 1ST DATA => HL  
; NUMBER OF PAGE => A  
; POINTER TO 1ST REGISTER ADDR. => IX

LOADRG: PUSH HL  
PUSH IX  
LD B,A  
SLA B ; 2X # OF PAGES &  
LDLOOP: LD C,(IX+O) ; RESET Z FLAG  
OUTI  
JR Z,LDEXIT  
INC IX  
JP LDLOOP ; NEXT  
LDEXIT: POP IX

3-12
Appendix A. Some Useful Subroutines (Continued)

POP     HL
RET

; ****************************
; **  SAVE PAGE REGISTERS  **
; **  SUBROUTINE            **
; ****************************

; THIS ROUTINE SAVES DATA OF PAGE REGISTERS INTO ARRAY
; POINTED BY HL. PASSED & RETURN PARAMETERS:
; NUMBER OF PAGES => A
; POINTER TO 1ST REG. ADDR. => IX
; POINTER TO 1ST SAVED DATA => HL

SAVREG: PUSH  HL
         PUSH  IX
         LD    B,A
         SLA   B ; 2X # OF PAGES &
SALOOP: LD    C,(IX+0) ; RESET Z FLAG
         INI
         JR    Z,SAEXIT
         INC   IX ; NEXT
         JP     SALOOP
SAEXIT: POP   IX
         POP   HL
         RET

; ****************************
; **   ERROR TRAP HANDLER   **
; ****************************

; THIS ROUTINE FINDS THE PAGE FAULT WHICH GENERATED NMI.
; PASSED PARAMETERS:
; REGISTER ADDRESS TABLE POINTER => IX
; RETURN PARAMETERS:
; FAULT DATA => DE
; REGISTER I/O ADR. LOW BYTE => C
; CAUSE => A (0 = INVALID ACCESS)
;               (1 = WRITE PROTECTED)

TRAP: IN    A,(3H) ; READ PORT O3H
       AND   OFH ; GOTCHA
       LD    B,O
       LD    C,A
       ADD   IX,BC
       LD    C,(IX+0) ; C HAS REG. ADDRESS
       IN    E,(C) ; READ LOW BYTE
       INC   C
       IN    D,(C) ; HI-BYTE
       DEC   C
       BIT   3,E ; TEST V BIT
       JR    Z,NVALID
       BIT   2,E ; TEST WP
       JR    NZ,WP
       LD    A,2 ; THIS SHOULDN'T
Appendix A. Some Useful Subroutines (Continued)

```
JP
NVALID: LD A, 0
JP
WP: LD A, 1
DONE: RET

; HAPPEN
; INVALID ACCESS
; WP PAGE
```
Increased speed, additional instructions and an addressing scheme that extends the available memory address space give the Z8108, an updated version of the Z80 microprocessor, greater flexibility.

On-chip memory management comes to 8-bit μP

The trend toward the use of high-level languages in microprocessor-based systems and toward complex configurations has created the need for more memory space, greater execution speed, easier access to software libraries, and in general, more sophisticated processor architectures. To those ends, the Z8108 is the first 8-bit microprocessor to provide on-chip memory management to expand memory addressing and a range of operating speeds of 6 to 25 MHz for increased throughput.

The initial member of the Z800 family, it is an enhanced version of the popular Z80 with new instructions and addressing modes for greater flexibility. In addition, a so-called system mode and a user mode of operation improve system reliability. The Z8108 also provides true 16-bit arithmetic capability and performs mathematical operations not done by the Z80.

The 40-pin chip includes a Z80-compatible bus interface with 8 address/data lines and 11 address lines, an on-chip clock oscillator, programmable dynamic memory refreshing, and expanded I/O addressing (Fig. 1). Because of its less stringent memory timing requirements, at an operating speed of 6 MHz the response time of the memories used need only be 250 ns. The processor's programmable-interrupt daisy-chain delay permits easy interfacing with most high-speed interrupt-driven devices; no external logic is required to generate additional wait states during an interrupt-acknowledgment sequence. Also, a large memory can be directly addressed without external bank-switching circuitry. Finally, because the processor executes all the instructions of the Z80, existing Z80 programs can be simply moved unchanged to the Z8108 for execution at increased throughput or easily modified to take advantage of the new processor's capabilities.

Looking at the architecture

Because the Z8108 is binary-code-compatible with the Z80, it has all the registers of the Z80, including dual 8-byte register banks A–L and A'–L'; two 16-bit index registers IX and IY; and a dual 16-bit stack pointer and program counter. One stack pointer is dedicated to system programs (including interrupts and traps), the other to user programs. The Z8108 has in addition a master status register that contains a number of flags to indicate the processor's current status. Also included are an interrupt and trap-vector table pointer and I/O page registers.

Programs on the Z8108 will be executed in either the system or the user mode. System programs have
access to all registers and instructions, but user programs are denied access to certain of these resources in order to provide a more secure environment—for example, one in which programs can be reserved in protected memory. The user mode is regarded as a subset of the Z80 instruction set because some Z80 instructions such as Halt are privileged in the Z8108 and can only be executed when the unit is in the system mode. Z80 programs will operate completely and correctly on a Z8108 since the processor assumes the system mode on power-up or reset.

The Z8108 addresses memory management in a number of ways. The on-chip memory management unit (MMU) maps system and user programs and instruction and data references separately, and easily remaps memory pages to different physical areas, thereby permitting easy access to very large physical memory spaces. Direct access to the memory management hardware is usually available only to system programs.

The Z8108's added instructions include some formalizations of undocumented Z80 instructions (such as accessing the index registers one byte at a time), in order to make the entire register set more orthogonal. Four new addressing modes increase the flexibility of the existing instructions and make code generation for high-level languages much easier. In addition, the Z8108 has a Test and Set instruction to provide synchronization for multiple processors, and both 8-bit and 16-bit multiplication and division instructions to increase throughput in computation-intensive applications.

The programmable bus timing feature increases system throughput. Control-bit settings allow the internal processor clock to be scaled for external bus accesses and wait states to be automatically inserted during bus cycles, as mentioned. Consequently, the user can select very high clock speeds to increase system performance without requiring high-speed memories and I/O devices.

The interrupt structure of the Z80 has been extended in the Z8108 to include program traps for exceptions and error conditions and a forced interrupt-service mode. This new mode provides automatic vectoring for each interrupt and trap, and provides support for nested interrupt processing.

With added interrupt-acknowledgment daisy-chain delay, the contents of a control register may be used to select a number of additional wait states to be added to interrupt-acknowledge cycles. Thus, slow peripheral devices or long interrupt daisy chains can be accommodated.

The Z80's input/output address space has been augmented in the Z8108 by the addition of the I/O page register that permits one of a number of blocks of I/O locations to be selected. Changing this register is a privileged operation that prevents any block from being accessed illegitimately.

The Z8108 includes an on-chip dynamic memory refresh controller. Refresh transactions can be enabled or disabled under program control and the refresh frequency can be selected. Unlike the Z80, the Z8108 generates separate bus transactions for refreshing, thus easing the memory-access timing requirements. Refresh cycles lost because of DMA-bus accesses or wait states are counted and automatically generated when the CPU regains control of the bus. The Z8108's refresh controller generates a 10-bit refresh address, ensuring support for very large dynamic RAM chips.

The on-chip oscillator-clock generator of the Z8108 simplifies system design by eliminating the need for an external MOS clock generator-driver. A crystal can be connected directly to the processor, or an external TTL-compatible clock signal can be provided. From this signal, the processor generates an internal clock, its frequency being one-half that of the input.

**Addressing modes**

Besides expanding the instruction set of the Z80 with four new addressing modes (see Table 1), the Z8108 extends some of the existing addressing modes (such as Register Indirect) to other instructions. The new modes are: Indexed with 16-bit Displacement, Stack Pointer Relative, Program Counter Relative, and Base Index.

![Diagram](image-url)

1. The 40-pin Z8108 microprocessor has a bus interface compatible with the Z80, an on-chip oscillator whose frequency is selectable from 6 to 25 MHz, and expandable I/O addressing. The Z8108 has all the registers of the Z80, plus a master status register, an interrupt and trap vector pointer, and an I/O page register for monitoring the processor's current status. The 16-bit microprocessor executes all software instructions of the Z80.
The Indexed with 16-bit Displacement mode is an extension of the Z80's Indexed addressing mode and uses a two-byte rather than a one-byte displacement. This method permits access to large dynamic data structures addressed by a pointer or access to arrays whose base address is known and whose index value can vary.

The Stack Pointer Relative mode is useful for high-level language applications where subroutine parameters and local variables are kept in the stack. Addresses of these variables are fixed offsets from the current top of the stack (located by the stack pointer) and therefore can be accessed directly using the Stack Pointer Relative mode.

With Program Counter Relative addressing, position-independent code—that is, code that uses only addresses relative to the current program location and not absolute addresses—can be produced. This procedure is useful for standard ROMs and subroutine libraries that can be loaded at different locations in memory for various applications, and it also reduces the time required to link-edit large programs. The Z80 has a few PC-relative instructions (all of them jumps), but the Z8108's PC-relative instructions include all the conditional jumps and calls, as well as 8-bit and 16-bit load, store, and arithmetic instructions.

Based Indexed addressing uses two registers to address an operand (any combination of the HL, IX, and IY registers may be used). The contents of the two are added to produce the effective address. In that way, both the base address of a structure and the index or offset can be computed at execution time (as is required for dynamic arrays). What's more, Base Indexing can be effectively combined with the other addressing modes, using the LDA (Load Address) instruction, to build up an arbitrarily complex addressing mode involving any combination of indexing and indirect addressing.

In addition to the new addressing modes, the old modes can be used for more instructions—for example, 16-bit Load and Store using the Register Indirect or Short Index mode, 16-bit ADD using an immediate operand, PUSH using an immediate value, and PUSH and POP using direct memory addressing (see Table 2). These extensions give the Z8108 the power and flexibility appropriate for both high-level and assembly language programming.

**More Instructions**

Foremost among the Z8108's new instructions are those for multiplication and division. The multiplication instruction has several variations, including an 8-bit-by-8-bit to 16-bit result and 16-bit-by-16-bit to 32-bit result with the operands addressable using any of the available addressing modes. Similarly, the division operations include 16-bit-by-8-bit to 8-bit quotient and remainder and 32-bit-by-16-bit to 16-bit quotient and remainder. The division instructions check for quotient overflow and attempted division by zero; these conditions will cause a trap, notifying the operating system to print a warning message or to abort the user program.

The Test and Set instruction has been included in

*2. The dynamic page relocator uses the processor's memory management unit to map and enable system and user programs independently. The Z8108's 16-bit logic addresses are divided into two fields for defining the physical addresses and for identifying the required set of page descriptor registers, one of which is used for system addresses, the other for user addresses. The state of the enabling flags determines which of the programs are serviced.*
the Z8108 to support multiprocessing. It tests the most significant bit of the operand, setting the condition codes appropriately and then sets the operand to all 1s. This primitive operation is often used as a signal between two or more cooperating programs to guarantee exclusive access while updating shared resources.

In addition to 16-bit multiplication and division, the Z8108's architecture includes other 16-bit arithmetic operations not found on the Z80. These instructions include 8-bit and 16-bit Sign-Extend, Add Accumulator to Addressing Register, 16-bit Compare, 16-bit Increment or Decrement in Memory, 16-bit Negate, and Full 16-bit Add and Subtract. All these operations use the HL register pair as a 16-bit accumulator.

The entire register set is more fully exploited in the Z8108 than in the Z80. The Z8108's IX and IY registers each can be accessed as a 16-bit register or as two single-byte registers (using any of the 8-bit load, store, or arithmetic operations). That capability in effect makes IX and IY into general-purpose registers like the BC, DE, and HL pairs.

The Z8108 architecture includes a new group of instructions for CPU control, to permit access to the new registers (such as I/O page and master status) and to handle system and user mode separation. The LDCTL (Load Control) instruction loads data into, or

---

**Table 1. The Z8108's addressing modes**

<table>
<thead>
<tr>
<th>Mode</th>
<th>Operand addressing</th>
<th>Operand value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>In the instruction</td>
<td>In a register</td>
</tr>
<tr>
<td>Register</td>
<td>Register address</td>
<td>Operand</td>
</tr>
<tr>
<td>Immediate</td>
<td>Operand</td>
<td></td>
</tr>
<tr>
<td>Register Indirect</td>
<td>Register address</td>
<td>Address</td>
</tr>
<tr>
<td>Direct Address</td>
<td>Address</td>
<td></td>
</tr>
<tr>
<td>Index</td>
<td>Register address</td>
<td>Index</td>
</tr>
<tr>
<td>Short Index</td>
<td>Register address</td>
<td>Address</td>
</tr>
<tr>
<td>Relative</td>
<td>Displacement</td>
<td>Pc value</td>
</tr>
<tr>
<td>Stack Pointer Relative</td>
<td>Displacement</td>
<td>Sp value</td>
</tr>
<tr>
<td>Base Index</td>
<td>Register address 1</td>
<td>Address</td>
</tr>
</tbody>
</table>

---
removes and stores data from the special CPU registers. Available only in the system mode, it is used to initialize the I/O page register and the interrupt and trap-vector table pointer.

A number of privileged instructions can be executed only by programs running in the system mode. These instructions provide control of the registers and processor state that transcend any one program and so are properly the province of the operating system. The privileged instructions include Halt, Enable, or Disable Interrupts, Select Interrupt Mode, Load the CPU Control Registers, and Return from Interrupts.

The SC (System Call) instruction provides an interface between user-mode programs and the operating system running in the system mode. A System Call pushes the processor status (in the program counter and master status register) onto the system stack, pushes a 16-bit system call number from the SC instruction onto the stack, and then executes a trap sequence. The operating system, after vectoring to the appropriate trap service routine, will normally use the system call number as an index into a table of subroutine addresses for the various system functions. This controlled mechanism lets user programs request privileged services such as memory management from the operating system without compromising the overall system and user protection mechanism.

One of the most troublesome problems of today’s microprocessor systems is management of large program and/or data spaces. This problem has been met in a variety of ways, such as adding external memory-mapping circuitry (increasing board space and complexity) and changing the design to use a 16-bit processor (losing compatibility with existing code and increasing development time).

### Memory space is quadrupled

The Z8108 tackles the problem by using the MMU to allow page-oriented memory mapping and provide protection without any external logic. The CPU itself separates system space from user space and program code from data references in both spaces, thereby quadrupling available memory space without changing existing program code or adding external hardware. An address translation mechanism, called dynamic page relocation, is then used to map these logical addresses into the physical address space. Logical addresses generated by the CPU are passed through the MMU and translated into physical addresses using this mechanism before being sent to the address lines coming out of a Z8108 chip.

Simply, the Z8108’s 16-bit logical address is divided into two fields, a 12-bit offset and a 4-bit index (Fig. 2). The offset is passed to the physical address unchanged, and the index selects one of the page descriptor registers. The indexed register contains the upper bits of the physical address and a set of so-called attributes for that page. These attributes indicate whether the table entry is valid (i.e., whether that page’s information resides in physical memory), whether writes are allowed to the page, and if so whether a write has actually occurred. If an access is attempted to a page marked as invalid, or a write is tried to a write-protected page, the instruction is aborted and a trap is taken. The system trap prevents a program from inadvertently accessing or modifying information not in its own purview.

As shown, the Z8108’s MMU actually contains two sets of page descriptor registers with separate enabling flags, one for system addresses, the other for user addresses. The appropriate set is chosen based on the state of the system/user flag in the master status register. Thus system and user programs can be independently mapped or unmapped, or mapped into different areas of physical memory. In addition, program and data separation can be enabled independently for each mode. If separation is enabled, the appropriate set of mapping registers is divided in half, with one half available for program accesses, and the other half for data accesses. In this case, only 3 bits of the logical address are used to select a page descriptor; the lower 13 bits of the logical

### Table 2. Addressing Comparison, Z80 vs Z8108

<table>
<thead>
<tr>
<th>Mode</th>
<th>Z80</th>
<th>Z8108</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack Pointer Relative</td>
<td>LD HL,nnn</td>
<td>LD A,(SP+nnn)</td>
<td></td>
</tr>
<tr>
<td>Base Index</td>
<td>PUSH HL</td>
<td>LD (HL)+X</td>
<td></td>
</tr>
<tr>
<td>Register Indirect</td>
<td>POP DE</td>
<td>ADD HL,DE</td>
<td>CALL (HL)</td>
</tr>
<tr>
<td>Index</td>
<td>PUSH HL</td>
<td>ADD HL,DE</td>
<td>CALL (HL)</td>
</tr>
<tr>
<td>Direct Address</td>
<td>LD HL(yyyyMMdd)</td>
<td>INC HL</td>
<td>INCW (yyyyMMdd)</td>
</tr>
<tr>
<td>Short Index</td>
<td>LD E,(X+24)</td>
<td>LD D,(X+25)</td>
<td></td>
</tr>
</tbody>
</table>

- approximates corresponding operation in Z8108
- equivalent operation
address pass through unchanged.

The Z8108 has a 512-kbyte physical address space. The 19 bits of physical address are produced by 12 or 13 bits from the logical address and 6 or 7 bits from the page descriptor registers. That translates into 128 pages of 4 kbytes each with program and data spaces integrated or 64 pages of 8 kbytes each with program and data references separated.

The processor provides a mechanism for system programs to access data using the user-mode mapping tables. Through the use of the LDUD (Load in User Data Space) and LDUP (Load in User Program Space) instructions, system routines can retrieve parameters from user programs (passed via the System Call instruction) or return values to user data structures.

The MMU registers of the processor are accessed by means of I/O instructions to a fixed set of port locations. These registers can be read or written singly or in blocks using the Z800 family’s block I/O instructions.

Using memory management

Using the memory management features is relatively simple. Since the MMU is part of the chip, no external logic is needed; the chip merely presents a large linear address range to the outside world. Simple Z80 programs running on a Z8108 need not worry about memory management, since the Z8108 powers up in the pass-through mode, which means that the logical address is passed directly to the physical address lines without translation.

Programs written especially for the Z8108 or Z80 programs that could benefit from a larger address space can use the memory management features in a variety of ways. The first technique is to separate the application program from the operating system. Thus both the application (running in the user mode) and the operating system (running in the system mode) can reside in different areas of physical memory, since they will use different sets of mapping registers. Second, the MMU can be set to separately map program and data references, allowing up to 64 kbytes of program code to access up to 64 kbytes of data (Fig. 3a).

If this technique does not provide enough addressing space, a variation of the bank-switching technique can be used (Fig. 3b). In this scheme, the program or data is broken into sections each 64 kbytes in length. As long as a program or data reference falls within the 64 kbyte range, normal addressing is used. But a reference to a different section must be preceded by a call to the operating system (using the System Call instruction) to change the page descriptor registers to map that reference. Either one page or the entire 64-kbyte address space can be remapped.

Another useful technique that takes advantage of the Z8108’s memory management is called virtual disk buffering. In this scheme, a large section of

---

Table 3: Recognition, Z80 vs Z8108

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Z80 opcode</th>
<th>Z8108 capability</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD</td>
<td>A,4OH</td>
<td>This is the proper operand.</td>
</tr>
<tr>
<td>DEFB</td>
<td>OCBH,03H</td>
<td>This is the key instruction.</td>
</tr>
<tr>
<td>JP</td>
<td>M,Z80</td>
<td>A Z80 will change the operand to</td>
</tr>
<tr>
<td>or</td>
<td>P,Z800</td>
<td>81H (shift left, insert 1), setting</td>
</tr>
<tr>
<td></td>
<td></td>
<td>the sign flag on the result.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A Z80 will test the original sign</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0) and clear the sign flag,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>then set A to all 1s.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Now test the flag and jump.</td>
</tr>
</tbody>
</table>

This instruction sequence exploits the difference; in one opcode between the Z80 and the Z800 family to allow a user program to decide which processor it is running on. The flags are set thus:

- **Inputs — none**
- **Outputs — Sign flag set according to CPU:**
  - S = 1 (M) if Z80
  - S = 0 (P) if Z800
- **Uses — A and F only**

The key instruction is in the one undefined shift group on the Z80 that actually performs a "logical shift left and insert 1" operation, with the same flag operation as the other shift/rotate instructions. This has been replaced on the Z800 with the Test and Set instruction that tests the sign of the operand, setting the sign flag accordingly, then setting the operand to all 1's. Thus with the proper choice of operand value, the sign flag resulting from this instruction becomes a Z80/Z8108 flag.
memory (typically 256 kbytes or more) is used to simulate all or part of a disk file. Whenever a disk block would normally be read into a memory buffer, the buffer is now simply mapped to point to the appropriate part of the virtual disk area. If this area is filled from the disk originally, all accesses to the file can be made to memory instead of to the disk, eliminating the long disk access times.

In summary, programs can now operate on large data bases in memory without using temporary disk files for storage. Programs larger than 64 kbytes can be run using the MMU to map different areas of the program in physical memory into the logical address space as they are needed. Cooperating programs running in a multitasking system can share portions of data memory, yet each can have private code and data that cannot be accessed by the other programs. These applications all rely on the simplicity and flexibility of the Z8108's paged memory management system and on the convenience of having the MMU as part of the chip.

The Z8108 also extends the I/O capabilities of the Z80. In addition to I/O transfers to and from registers, data to be sent or loaded can be transferred directly to or from memory. That gives greater flexibility in I/O transfers and can result in greater throughput to the external device. The architecture also has the Z80's block input and output instructions for even greater I/O transfer rates.

Also, the I/O addressing space of a Z8108 is larger than that of the Z80. The content of the special I/O page register is used to drive the upper address bits during an I/O transaction, thereby permitting banks of ports to be selected. The Z8108 supports eight banks of port locations within the I/O address space. Because input and output themselves need not be privileged operations in the Z8108, the I/O page mechanism affords protection to critical devices (such as the on-board MMU) on a page basis, since access to the I/O page register is always a privileged operation.

**Interrupts and traps**

The three interrupt service modes of the Z80 have been expanded in the Z8108 by the addition of a fourth mode and by the addition of internal interrupts or traps using this mechanism. The four interrupts are modes 0 to 3, with modes 0, 1, and 2 operating in the same way as in the Z80. Mode 0 expects an instruction to be placed on the data bus during the interrupt acknowledgment cycle that is executed to begin the interrupt service routine. Mode 1 ignores the data and executes an unconditional jump to location 0038H. Mode 2 uses the contents

3. Separately mapped program and data references double the Z8108's addressing space. Eight descriptor registers are used to map program addresses, and eight to map data addresses (a). Switching between banks of data can be done simply by changing the eight data-page descriptor registers to a new block of physical memory (b).
of the special I register, along with the data read during acknowledgment, to point into a table of subroutine addresses, which dispatch the service routine. Interrupt Mode 3 uses the interrupt and trap vector table pointer register to point to an array of new program status values (each consisting of a new program counter value and a new master status register value) for the traps and nonvected interrupts and an array of new program counter values for use with vectored interrupts.

If a vectored interrupt is accepted in mode 3, the old contents of the program counter and the master status register are saved on the system stack and an interrupt vector is read from the interrupting device. This value is then saved on the system stack and used to fetch new contents for the program counter from the trap vector table. This sequence allows an interrupt to vector to any location in memory for service and also permits complete nesting of interrupts, since the previous state of the interrupt enable is saved on the stack, not just in a temporary flag register as in the Z80.

The processor supports both maskable and nonmaskable interrupts. Maskable interrupts are enabled by a bit in the master status register and are accepted only if the bit is set. Nonmaskable interrupts cannot be disabled and are always accepted. The processor checks the state of the external interrupt pins at the end of the current instruction (or the end of an iteration of one of the block instructions) and executes the interrupt service sequence before continuing with the next instruction. Maskable interrupts can be accepted as either vector or nonvector. If they are to be vectored, processing occurs as described above. If nonvector (and in interrupt mode 3), a special nonvector interrupt table entry is used to dispatch the interrupt service routine.

Traps in use interrupt mode 3 to vector to a service routine and to load a new master status value for that routine. Thus a trap can be at least partially serviced in a user-mode program. The Z8108's traps include Privileged Instruction, System Call, Page Fault (from the MMU), Division Exception, Single

4. A system using the Z8108 may be designed into an existing system using the Z80, peripherals, and medium-speed memory devices. Having multiplexed address and data buses and an internal oscillator, the processor cuts the package pin count without reducing flexibility.
Step, and Breakpoint on Halt. The last two facilitate program debugging by providing a reliable means of stepping through programs one instruction at a time and breaking program execution at any instruction, respectively.

Following power-up or a reset, the Z8108 will behave like a Z80 (or an 8080). This means that memory management is disabled, the system/user flag is set to system (allowing all privileged instructions to be executed), the system stack pointer is enabled, the I/O page register is cleared, and the interrupt response is set to mode 0. All the Z80's instructions run identically on the Z8108. The Z8108, however, operates two to eight times faster.

But what if a program needs to know whether it is running on a Z80 or on a Z8108 (in order to take advantage of the Z8108's power if it runs on one but still be capable of execution on a Z80)? One of the new instructions in the Z8108 replaces a previously undocumented instruction of the Z80, permitting a program to determine which processor it is running on. The program achieves this by performing a test sequence on the new instruction (see Table 3). The instruction sequence is used to skip the initialization procedure needed to activate the Z8108 if the program is running on a Z80 or to jump to in-line Z8108 code (to do a multiplication, for instance) rather than using a Z80 subroutine for the function.

Designing a system

The Z8108 has a multiplexed address and data bus to reduce the package pin count without sacrificing performance (memory transactions still require only three clock cycles). In addition, design with the Z8108 is easy because of the on-chip oscillator, memory refresh mechanism, and programmable bus timing features. Figure 4 shows an example of a Z8108 design using existing peripherals and medium-speed memory devices.

Note that the only external element required in the oscillator circuit is a crystal (whose frequency is twice the desired internal frequency). The external clock output (CLK) line provides a system clock at the internal clock frequency divided by the programmable bus timing value. The multiplexed address and data bus is easily demultiplexed with a standard low-power Schottky 8-bit latch. The Address Strobe (AS) signal is used to gate the address into the latch. The rest of the signals generated by the Z8108 are compatible with standard Z80 signals.□
An advanced microprocessor family adds on-chip cache and memory management yet retains software compatibility with its predecessor. It gives the designer a virtual mainframe on a chip.

8- and 16-bit processor family keeps pace with fast RAMs

For years, designers have not been able to take full advantage of the speed of available RAMs. In otherwise efficient microcomputer setups, the processors have been the main drag on throughput. This situation will change shortly with the introduction of a new family of 8- and 16-bit processors. These successors to the popular Z80 microprocessor are expected to operate at a 25-MHz clock frequency and can use a burst mode on their 16-bit bus to work with 80-ns RAMs. But that is not all.

The Z800 family, to be fabricated using an advanced NMOS process, will have on a single chip such features as a cache memory, memory management, counter-timers, DMA controllers, and serial I/O. Add to that new instructions to ease software development and the designer will have a virtual mainframe at his disposal.

The family consists of four members, two with an 8-bit, Z80-compatible interface and two with a 16-bit, Z-bus (Z8000 family) interface. All members are totally code-compatible with the Z80 microprocessor. The new instructions, combined with the on-chip resources and high clock rate, extend performance to the 5-million-instructions/s level, as simulated via a Pascal compiler. This rate is competitive with many of the so-called 32-bit microprocessors.

To achieve the high clock rate, a 2-μm n-channel process was used. There are two levels of polysilicon interconnections, the first a low-resistance layer and the second for interconnections and high-impedance load resistors. The process incorporates four transistor types, as defined by their thresholds: one enhancement, one intrinsic, and two depletion-mode devices.

The members of the Z800 family consist of the 8-bit Z8108 and Z8208 and the 16-bit Z8116 and Z8216 (see Table 1). However, only the Z8208 and Z8216 have the on-chip peripherals and a full 16-Mbyte address space. To reduce the board space, these processors are housed in dual in-line packages with pins on 70-mil centers, permitting a 64-pin package to fit in the board area of a 48-pin DIP having leads on 100-mil centers.

With the Z-bus interface, the processors offer twice the system throughput of the 8-bit bus devices. They can take advantage of all the Z-bus peripherals already available for the Z8000 family of 16-bit processors.

The architecture of the Z800 processor core resembles that of the Z80 microprocessor, with the addition of several registers to increase flexibility. As part of the architectural enhancements, the processor has been set up to operate in either a system or a user mode. In the system mode, all of the instructions can be executed and all of the CPU registers accessed. This mode may be used with programs that perform operating system functions, and it can also run Z80 software emulation. In the user mode, some instructions cannot be executed and some CPU registers are made inaccessible. Thus, system integrity is ensured, even by run-away application software that might otherwise alter operating system information.

Enhanced instruction set

Supporting the two modes are two stack pointers, one for the system mode and one for the user mode. Additional flexibility was added to the register set by the high- and low-order byte addressability of the 16-bit IX and IY index registers.

The instruction set contains all of the Z80 commands, and then some. Added are 8- and 16-bit multiplication and division operations; Sign Extend,
16-bit Compare, Negate, and Increment and Decrement in Memory; System Call; test and set commands; several load control instructions; and some commands that interface with the extended processing units, such as the forthcoming Z8070 floating-point math processor.

Multiprocessing is supported by the Test and Set instructions, which facilitate communication between programs that share resources. The Load Control instruction group is used in the system mode to set up registers that configure on-chip resources and to poll the chip status. The System Call instruction enables User programs to request services available only in the processor's system mode—the enabling or disabling of interrupts, for example.

Abundant silicon resources

Along with the new instructions come four new addressing modes: index, base-index, stack-pointer-relative, and program-counter-relative. These are in addition to the five modes carried over from the Z80 (register, immediate, direct-access, register-indirect, and short-index).

An abundance of on-chip resources is available for the designer (Fig. 1). The Z8216, the most complex member of the family, and the 8208 have the Memory Management Unit, cache memory, four 16-bit counter-timers, a serial port, four channels of DMA control, and a dynamic RAM refresh controller. These on-chip peripherals can also be linked internally for further enhancement of their capabilities. However, even the 40-pin Z8108 and Z8208 have the four counter-timers available for internal timer applications.

The on-chip memory manager coordinates the 16-Mbyte address space of the Z8208 and Z8216 processors (Electronic Design, Oct. 14, 1982, p. 163) with no speed penalty during the address translation. On the Z8108 and Z8116, 19 address lines provide access to 512 kbytes of memory. To translate between the logical and physical address spaces, the memory manager uses two sets of 16 page-descriptor registers—one set for the system

---

1. The high-end member of the Z800 family, the Z8216, has on-chip resources that give it the characteristics of a full minicomputer. Included are a memory management unit, a cache memory, multiple DMA channels, multiple counter-timers, and a serial port.
mode and one for the user mode. Each 16-bit page descriptor register contains 12 bits of address information and 4 bits of attribute information.

Addresses are translated when the lower 12 or 13 bits (depending on whether the program/data separation option is enabled or disabled) of the logical address is concatenated to the address information contained in the appropriate page descriptor register (Fig. 2). This register is selected by the most significant bits in the logical address.

Attribute bits control access and provide status information for each page. They include a Valid bit, which indicates whether or not a page descriptor is valid for use; a Write Protect bit, which permits a page of memory to be read only; a Modified bit, which indicates whether a page in memory has been written to; and a Cachable bit, which indicates whether a page may be loaded into the cache memory. The combination of the Modified bit and the ability to abort and restart an instruction upon an access violation thus permits the processor to implement a virtual memory system.

To improve the access time for often-used or time-critical program sections, an on-chip cache memory consisting of 256 bytes is included on all Z800 processors. This cache can be configured to be instruction-only, data-only, or a combination of both. Since this memory is on the chip, no speed penalty is incurred when stored items are accessed.

Operating on the principle that recently used instructions or data have a high probability of being called up again, the cache holds the most recently accessed code, thereby permitting repetitive items to be executed much faster. Every time the processor requires data or an instruction, it first checks the cache memory to see if the item is present. If it is, the processor will use it, and no external bus access will be made. It is estimated that the use of the Z800's cache memory, will make the execution of Z80 code some two to eight times faster.

Inside the cache memory

When configured as a cache, the memory is organized into 16 lines of 16 bytes each (see Table 2). Associated with each line are two fields—a 20-bit physical address tag and a 16-bit “valid” field. The address tag is matched against the most significant 20 bits of every physical address generated by the CPU and the memory manager, and if a match is detected on any of the 16 tag addresses, the lower 4 bits of the physical address are used to select the appropriate byte or word in the matched line. The valid field contains one Valid bit corresponding to each byte in the line.

If the appropriate Valid bit for the byte accessed in the matched line is set, a cache “hit” occurs, and that byte is used by the CPU. If the bit is not set, the processor sends the address to the external memory to fetch the data. This data is then used by the processor and written into the cache, which causes the Valid bit to be set for each byte written into the cache. If none of the 16 tag addresses match the

Table 1. How the members of the Z800 family line up

<table>
<thead>
<tr>
<th>Package</th>
<th>Data bus interface (bits)</th>
<th>On-chip peripherals</th>
<th>Common features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z8108</td>
<td>40</td>
<td>8</td>
<td>Four 16-bit counter-timers (internal only)</td>
</tr>
<tr>
<td>Z8116</td>
<td>40</td>
<td>16</td>
<td>Four 16-bit counter-timers (one internal only)</td>
</tr>
<tr>
<td>Z8208</td>
<td>64</td>
<td>8</td>
<td>Four 16-bit counter-timers (one internal only)</td>
</tr>
<tr>
<td>Z8216</td>
<td>64</td>
<td>16</td>
<td>Four 16-bit counter-timers (one internal only)</td>
</tr>
</tbody>
</table>

Advanced processor family

20-bit address, the line in the cache that has been used least recently is "flushed"—that is, the processor clears all the valid bits to invalidate the bytes—and the 20-bit address becomes the new tag address. The appropriate byte or bytes are then pulled from the external memory.

The Z-bus interface on the Z8116 and Z8216 permits the processors to use a burst-mode bus transaction to preload the cache. Although the burst mode was designed for use with the new 64-kbit dynamic RAMs that support a serial nibble output, it will also work well to fill up the cache memory.

If the cache memory is not needed, the circuitry can be disabled and the memory reconfigured as 256 bytes of fixed-address RAM. This "local" memory can be used with ROM-only systems, or it can hold those portions of a program that need the speed of on-chip memory, such as interrupt routines. In the fixed-address mode, the tag addressed identify individual lines, but the settings of the Valid bits have no meaning. Tag addresses can be set by the programmer and will remain fixed to guarantee the addresses of the memory.

On-chip peripherals add power

With their ample peripherals on the chip, Z800 microprocessors are, in effect, full systems on a minimum of board space, with minimum device interconnections and components. They are excellent for cost-sensitive applications. The four DMA channels of the Z8208 and Z8216 provide independent, high-speed data transfers; the serial port, a full-duplex asynchronous interface capable of operating at up to 2 Mbits/s at a 10-MHz clock rate. Each of the DMA channels can be programmed to transfer data from memory to memory, from memory to an I/O device (or vice versa), or from one I/O device to another. Moreover, data can be transferred in any of three modes: single-transaction, burst, or continuous.

In the single-transaction mode, the DMA section releases the bus to the CPU or another DMA channel between each byte or word transfer; the burst mode permits the DMA section to transfer data as long as the requesting peripheral remains ready. The continuous mode, on the other hand, allows the DMA circuit to transfer an entire block of data without releasing the bus. Also, each channel of the controller can operate in a "no transfer" mode, in which it acts as a counter.

Each DMA channel consists of a 24-bit source address register, a 24-bit destination address register, a 16-bit count register, and a 16-bit transfer descriptor register. All these registers are in the I/O space of the CPU and are accessed with the word I/O instructions over the CPU's internal bus.

Externally, the DMA channels use the address, data and control lines of the processor to transfer the data. Each channel has an input pin associated with it, to notify the channel that an external device is requesting a transfer.

Controlling all four channels is a master DMA control register that can direct the channels to link with one another or to the serial I/O channel. When DMA channels are linked, one channel acts as a slave that loads the master with new address, count, and descriptor information. The master channel transfers a block of data to the destination and then waits while the slave updates its registers from in-

| Table 2. How the Z800's cache memory is organized |
|-----------------------------|-----------------|-----------------|-----------------|
| Line 0 | Tag 0 | Valid bits | Cache data |
| Line 1 | Tag 1 | Valid bits | Cache data |
| Line 2 | Tag 2 | Valid bits | Cache data |
| ... | ... | ... | ... |
| Line 15 | Tag 15 | Valid bits | Cache data |

3. Linked DMA operations can be set up with two of the on-chip DMA channels. One channel can be used to download control information to another channel, thus minimizing the number of times the processor must stop to transfer control parameters.
formation transferred from memory (Fig. 3). With this structure, transfers of different types and to different locations can be initiated without CPU intervention.

Although all the processors have four counter-timers on chip, only the Z8208 and Z8216 take the lines of three to the outside; the fourth counter-timer is an internal-only function on all four devices. However, the three externally available counter-timers on the Z8208 and Z8216 are full 16-bit down counters that can be independently programmed to count external events (count mode) or internal clock cycles (timer mode). Two of the 16-bit counters also can be internally linked to form a 32-bit counter.

In use, each counter is loaded with an initial value that is also latched into the 16-bit time-constant register of that counter. When the counter value reaches zero, the counter causes one of several things to happen: an interrupt is generated, an external pulse is generated, or the counter is reloaded from the time-constant register to restart the countdown sequence. Command bit options specify which of those events occurs. In addition, each counter can be gated or triggered by either external signals or software, thus providing an extra measure of control.

**Serial port shines**

The serial port usually takes advantage of one of the timers as a baud-rate generator or an external clock source. The serial port can send and receive data simultaneously, and two of the DMA channels can be linked with the transmitting and receiving sections to provide automatic high-speed serial transfers. Like most universal asynchronous receiver-transmitters, the port handles a data format that consists of a start bit; five to eight data bits; even, odd, or no parity; and one or two stop bits.

The serial port also can be used to load data or programs remotely if a Z800 device is used as a slave to a larger host system. This remote-loading capability is supported by a bootstrap mode that can be selected when the processor is reset. When selected, this mode automatically links a DMA channel to the receiver side of the serial port, programs a default destination (000000) into the DMA channel, sets up the serial port data format, and begins loading 256 bytes of data into memory via the serial channel. That permits the Z800 to serve as a ROM-less slave processor, subject to changes to suit the needs of the host system.

**Multiprocessor operation made easy**

Besides serving as slave processors, the Z800 units can operate in multiprocessor systems. Both the Z8208 and the Z8216 have on-chip features that readily permit their incorporation into multiprocessor systems.

In the example (Fig. 4), two or more processors, each with a local bus that supports some combination of memory and I/O devices, communicate via a memory block on the shared global bus. This architecture requires the use of bus arbitration logic to allocate the global bus resource.

Only part of each Z800's address space would be assigned to the global bus via the processor's local-address register. Included in this scheme could also be a master processor to control the global bus and
A complete microcomputer system can be built around the Z8216, because its powerful resources eliminate many peripheral functions. For parallel I/O and interrupt control, two Z8036s can be added, and a Z8030 serial communication controller can add two more serial I/O channels.

allocate tasks to the slave Z800 processors.

For maximizing board space for memory, the Z8216 is the best choice. It offers many of the functions a designer needs to build a microcomputer board. All that must be added are the interface logic and buffers required to tie into a system bus like the IEEE-696 or IEEE-796.

To handle interrupts and provide a parallel port for a printer, two Z8036 counter-timer and parallel I/O circuits can be added. For additional serial I/O, a Z8030 dual-channel serial communications controller can be connected to the local bus (Fig. 5).

Since the processor contains its own clock oscillator as well as a clock output, all timing can originate from its crystal. One of the counter-timers acts as a baud-rate generator for the built-in serial port, and the off-chip serial communications controller has its own baud-rate generator, reducing system complexity.

The special status and control signals available from the Z8216 simplify the external logic needed to generate the bus and buffer control signals. To demultiplex the lower 16 address/data lines, the address latch must simply be strobed with the address strobe line, and the status lines can readily be decoded by either a 1-of-10 or a 1-of-16 decoder. (The first 10 status outputs are used in systems that do not have an extended processing unit, so the smaller decoder can be used. If an extended processing unit is present, the remaining six outputs should be decoded.)

Since the processor contains its own 10-bit refresh-address generator, dynamic RAMs as large as 1 Mbit can readily be handled without the space-consuming refresh logic often needed in medium-size systems. Also, the processor can automatically generate the appropriate wait states, thus permitting the bus timing to be optimized for the memory access speed.

Acknowledgments
The authors would like to thank Greg Barr, Gary Cole, Monte Dalrymple, Khue Duong, Bob Kurihara, Stanley Lai, Donald Mar, Lan Nguyen, Mike Pitcher, Gurdev Singh, and Irving Stuart for their valuable contributions to the development of the Z800 processors.

<table>
<thead>
<tr>
<th>How useful?</th>
<th>Circle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate design application</td>
<td>556</td>
</tr>
<tr>
<td>Within the next year</td>
<td>557</td>
</tr>
<tr>
<td>Not applicable</td>
<td>558</td>
</tr>
</tbody>
</table>

Electronic Design • April 28, 1983
Z8000™ 16-Bit Microprocessor Family 4

Zilog
Cost-Effective Memory Selection for Z8000™ CPUs

February 1982

The "memory-effective" architecture of the Z8000 CPU is the key to cost-effective system design in many applications. Z8000 CPUs are designed to achieve high performance without the use of high-performance memories. Because a single application often requires hundreds of memory chips for each CPU, this memory-effective design can result in large cost savings.

Many factors enter into the selection of CPU and memory characteristics for a given application. This application note examines the simple formula that relates these factors to each other and provides examples of the formula applied in common situations. Background for the material in this application note can be found in the Z8000 CPU Manual (document #00-2010-C0) and in the Z8001/Z8002 CPU Product Specification (document #00-2045-A0).

THE BASIC FORMULA

Figure 1 shows a generalized view of the information path taken when the CPU issues a valid memory address. This process ends when valid data, representing the contents of the addressed location is returned to the CPU. Not all of the elements shown in Figure 1 are necessarily present in every application, in which case the basic formula is simplified for that application.

This schematic view shows the principal elements that enter into the basic formula relating memory and CPU timing characteristics. Many applications use subsets of these elements, which simplifies the basic formula for those applications.

The two-letter symbol in each box is used in the basic formula to represent the time length of that box's task.

Figure 1. The Address-to-Data Path Illustrates the Basic Formula
The address issued by the CPU is called a logical address. It is transformed by the MMU (or other memory management circuitry) into a physical address. The symbol "MM" in Figure 1 represents the time required for this transformation. When no address translation circuitry is present in a given application, MM=0.

When a physical address is emitted by the MMU (or by the CPU if address translation is not used), it is presented to the memory array. After an interval of time represented by "MA" in the basic formula, data representing the contents of the addressed location appear at the output of the memory. If no error check/correction circuitry is used in a given application, then no check bits appear, and the output of the memory is presented to the CPU as valid data representing the contents of the addressed location. If error correction circuitry is used, then the memory output is input to the error check/correction circuitry. After an interval of time represented by EC in the basic formula, the output of the error check/correction circuitry is presented to the CPU as the contents of the addressed location.

The three time periods represented by MM, MA, and EC all contribute to the total time elapsed in the address-to-data path, but one additional calculation is required to reach the total. MM, MA, and EC represent the times elapsed in the corresponding elements in the information path. The remaining term, BD, represents the time elapsed while passing information between the specific areas. Thus, BD must include the delays in any buffers required for interboard bus transfers and time spent in address decoders or other selection logic. Even the time taken for propagation of signals must be considered, although the amount is usually negligible in comparison with MM + MA + EC.

The total time elapsed in the address-to-data path is the sum of the four terms MM, MA, EC, and BD. This total must be less than the maximum, CD, specified for the given CPU. This leads to the most fundamental form of the basic formula:

\[ MM + MA + EC + BD < CD \]  \hspace{1cm} (1)

The term CD, however, can also be expressed as a formula. CD depends partly upon the characteristics of the clock supplied to the CPU and partly upon constants that depend upon the maximum clock speed rating of the CPU. Furthermore, the Z8000 architecture allows "wait states" to be inserted into memory access transactions. The number of wait states inserted is another factor entering into the formula for CD. Finally, there are two possible expressions for CD, depending upon whether independent timing or the address strobe signal (AS) is used to signal "address valid."

The published ac characteristics of the Z8000 CPUs specify the exact point at which addresses become valid. (Parameter 9 of the ac characteristics table relates this point to a rising clock edge.) An address strobe signal, AS, is also provided by the Z8000 CPU. The rising edge of AS, which occurs approximately one-half clock period after addresses become valid, can be used to signal "address valid." Use of AS simplifies the circuitry but places a greater demand on the memory. Furthermore, no similar signal is available from the MMU circuits designed for use with the Z8000 CPUs, so that AS can only be used as described above in a system without memory address translation (i.e., when MM=0).

The two ways of computing CD (ac characteristic parameters 11 and 27) are expressed in the following two equations:

\[ CD = (2+W) \cdot CP + CH - K1 \]  \hspace{1cm} (2a)

\[ CD = (2+W) \cdot CP - CF - K2 \]  \hspace{1cm} (2b)

where:

- \( W \) = number of wait states
- \( CP \) = clock period
- \( CH \) = clock width (high)
- \( CF \) = clock falling time
- \( K1, K2 \) = constants whose values depend on the rated maximum clock speed of the CPU.

The right hand side of equation (2a) expresses the time between the actual appearance of a valid address output and the point at which valid data is required. The right hand side of equation (2b) expresses the time between the rising edge of AS and the point at which valid data is required. The values of K1 and K2 for Z8000 CPUs are given in Table 1.

The foregoing considerations can now be summarized in the basic formula (Figure 2). There are two versions of this formula, one for each of the two expressions for calculating CD (2a and 2b).
### The Basic Formula

**(Two Versions)**

\[
MA < (2+W) \cdot CP + CH - (MM + EC + BD + K1) \quad (A)
\]

\[
MA < (2+W) \cdot CP - CF - (EC + BD + K2) \quad (B)
\]

- \( MA \) = rated access time of the memory
- \( W \) = number of wait states
- \( CP \) = clock period
- \( CH \) = clock width (high)
- \( CF \) = clock fall time
- \( MM \) = memory translation (MMU) overhead
- \( EC \) = error check/correction overhead
- \( BD \) = selection logic, buffers, bus delay
- \( K1,K2 \) = constants (see Table 1)

The basic formula determines the maximum access time for memories used with a Z8000 CPU as a function of any factors that might affect it. The first version of the formula is the general case and assumes that an independent circuit is used to signal the memory when the CPU or the MMU emits a valid address. The second version, not applicable if memory management is used, assumes that the rising edge of address strobe (AS) will be used to generate the RAS or equivalent signal to the memory.

#### Table 1. CPU Speed Rating Affects the Basic Formula

<table>
<thead>
<tr>
<th>Maximum Rated Clock Speed</th>
<th>4 MHz</th>
<th>6 MHz</th>
<th>10 MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>K1</td>
<td>130 ns</td>
<td>95 ns</td>
<td>60 ns</td>
</tr>
<tr>
<td>K2</td>
<td>120 ns</td>
<td>100 ns</td>
<td>50 ns</td>
</tr>
</tbody>
</table>

**THE WAIT STATE TRADEOFF**

As either version of the basic formula shows, adding a wait state to the process increases the maximum memory access rating (MA) by one clock period (CP). (Fractions of wait states can be simulated by "clock stretching," to which the discussion in this section also applies.) CPU performance, however, is lessened by the introduction of wait states. This section is concerned with the estimation of that reduction.

The decline in performance level attributable to the introduction of wait states into memory accesses is difficult to pinpoint, since each instruction is affected differently. For example, a register-to-register multiplication takes 70 clock periods without wait states and 71 clock periods with a wait state—a reduction of 1.4% in execution speed. A register-to-register load, on the other hand, takes three clock periods without...
wait states and four clock periods with a wait state—a reduction of 25% in execution speed.

In one published study (AMD, Z8000 Benchmark Report, 1981), five Z8000 programs were analysed. The objective was to compare Z8000 performance with that of competing microprocessors, but included in the reported results was a performance comparison of each of the five Z8000 programs with and without a wait state. The reductions in execution speed were 5%, 6%, 15%, 17% and 21%. The 5% and 6% reductions appeared in the "automated parts inspection" and "XY transformation," both of which involve many register-to-register arithmetic operations and few memory reference instructions. The 15% and 17% reductions appeared in the "block translation" and in the "bubble sort," both of which involve a great many memory accesses. The 21% reduction appeared in a dummy "reentrant procedure," which does almost nothing other than save and restore the general registers.

As the study cited above shows, the effect of adding wait states varies from application to application. If a numerical value can be assigned to the reduction in performance level caused by wait states in a given application, then that value can also be compared with the reductions arising from other approaches to providing a given target memory access rating, such as:

• Reducing the clock speed (increasing CP).
• Using values of W other than 1.

The effect of each of these alternatives can be evaluated numerically and compared with the effect of adding one wait state.

Reducing Clock Speed

Assume that values have been assigned to all of the variables in the basic formula and that wait states are desired to achieve a higher upper bound on MA. If ΔMA is the desired increase in the right side of the basic formula, then each version of the basic formula gives rise to an equation for the required change ΔCP:

\[
\begin{align*}
\Delta CP &= \frac{\Delta MA}{2 + W + CH/CP} \quad (3a) \\
\Delta CP &= \frac{\Delta MA}{2 + W} \quad (3b)
\end{align*}
\]

Since the execution speed of the CPU is inversely proportional to the clock period, the ratio of the new speed to the old after the change ΔCP in clock period is

\[
\begin{align*}
p &= \frac{CP}{CP + \Delta CP} = \left(1 + \frac{\Delta MA}{(2+W)\cdot CP+CH}\right)^{-1} \quad (4a) \\
p &= \frac{CP}{CP + \Delta CP} = \left(1 + \frac{\Delta MA}{(2+W)\cdot CP}\right)^{-1} \quad (4b)
\end{align*}
\]

For example, assume that version (B) of the basic formula has been used with values \(W = 0\), \(CP = 250\)ns (4 MHz), \(CF = 10\)ns, \(EC = 0\), \(BD = 60\)ns, and \(K2 = 120\)ns. Then \(MA < 500 - 10 - (60 + 120) = 310\)ns. If memories rated at 350ns access time are desired the required \(\Delta MA\) is 40ns. Using (3b), the required \(\Delta CP\) is 20ns, leading to a new CP of 270ns, which corresponds to a clock speed of 3.70 MHz. Formula (4b) gives a value of

\[
p = \left(\frac{40}{1 + 500}\right)^{-1} = .92
\]

That is, reducing the clock speed to achieve the desired memory access time results in an 8% reduction in execution speed. If, instead, one wait state had been inserted (increasing the maximum MA from 310ns to 560ns), the reductions in execution speed for the programs cited above would range from 5% to 21%.

Using Values of W Other Than 1

Assume that values have been assigned to all of the variables in the basic formula and that wait states are desired to achieve a higher upper bound on MA. Assume also that a relative performance level of \(p\) is achieved when \(W = 1\). (For example, for the five programs cited earlier, the values of \(p\) would be .95, .94, .95, .83, and .79.) Then, for either version of the basic formula, the performance level corresponding to \(W\) wait states is given by

\[
p = \frac{p_0}{p_0 + (1 - p_0)\cdot W} \quad (5)
\]
Thus, for example, if insertion of one wait state leads to a performance level of .85 (a reduction of 15%), the insertion of one-half wait state (by clock stretching) leads to a performance level of

\[ P = \frac{.85}{.85 + (.15)(.5)} = .92 \]

or a reduction of 8%.

**EXAMPLE 1: THE ZILOG SYSTEM 8000**

The Zilog System 8000 provides an example that includes all of the elements of the basic formula. The following characteristics describe the main memory of the System 8000:

- **MA** = 150ns (dynamic RAM)
- **W** = 0
- **CP** = 180ns (5.56 MHz)
- **CH** = 80ns
- **MM** = 90ns (Z8010 MMU, 6MHz rated)
- **EC** = 40
- **BD** = 60 (Buffers and selection logic)
- **K1** = 95ns (Z8001, 6 MHz rated)

Version (A) of the basic formula must hold:

\[ 150 < (2+4)\cdot180+80-(90+40+60+95) = 155 \]

The difference of only 5 ns indicates that the system characteristics have been closely matched. Notice that the clock is running at less than the rated maximum speed. An increase to the maximum allowed for a 6 MHz Z8001 CPU would result in a clock period (CP) of 165ns, and thus a maximum memory access rating (MA) of 118. The 5.56 MHz clock speed results in a relative performance level of 165/180 = .92, or an 8% reduction in execution speed.

**EXAMPLE 2: A Z8002 WITH A Z6132**

The Z6132 quasistatic 4K byte RAM is designed for use with the Z8000 CPUs. For example, with the Z8002's AS line tied directly to the AC input of the Z6132 (see Figure 6 of the Z6132 Product Specification, document number 00-2028-A0, version (B) of the basic formula can be used:

\[ MA < 2\cdot CP - CF - K2 \]

For 4 and 6 MHz rated CPUs running at maximum speed and using the longest allowed clock fall time (ac characteristic parameter 4), the basic formula gives:

\[ MA < 2\cdot250 - 140 = 360 \text{ ns} \quad (4 \text{ MHz}) \]
\[ MA < 2\cdot165 - 110 = 220 \text{ ns} \quad (6 \text{ MHz}) \]

Thus, a 350ns Z6132 can be used with a 4 MHz Z8000 and a 200ns Z6132 can be used with a 6 MHz Z8000.
These benchmarks compare the performance of the Z8001 and Z8002, the Motorola 68000 and the Intel 8086 running the set of programs which have become industry standards for comparing microprocessors. The data demonstrates that:

- The 6MHz Z8000 outperforms the 8MHz 68000 and any version of the 8086.
- At any given memory access time, the Z8000 gives higher performance than the 8086 or 68000.
- Any given performance level can be reached with the Z8000 using slower memories than the 8086 or 68000.

For a demanding microprocessor application the user has the choice of three competing microprocessor families:

- The Z8000 manufactured by Zilog and AMD
- The 8086 (or iAPX 86/10) manufactured by Intel
- The 68000 manufactured by Motorola

A widely quoted benchmark comparison of these three microprocessors was published by Intel in 1980 under the title "16-bit Benchmark Report iAPX86, Z8000 and 68000" (Intel Publication No AFN01551A).

Not surprisingly, the Intel 8086 was announced the winner in that publication. Intel achieved this result by inefficiently coding the competing devices, thus not utilizing the powerful instruction sets of the more modern Z8000 and 68000 microprocessors.

In order to refute the wrong conclusions drawn by Intel, we purposely used the same benchmarks, and even the identical flow diagrams. We give Intel the benefit of the doubt and assumed their performance figures from the above mentioned document. For the Z8000 and the 68000, however, we rewrote the code efficiently. We did not use exotic tricks, just plain straightforward, efficient coding that takes advantage of the powerful instructions of the Z8000 and the 68000.

We made one minor modification to the Intel definition of the Block Translation. We write the translated character back into the same buffer where the EBCDIC character was stored. We see no reason why anybody would perform a non-destructive translation. It wastes memory space. The purist who wants our exact response to the Intel benchmark should subtract 13% from the Z8000 performance to accommodate non-destructive translation, which happens to be less efficient on the Z8000, but does not affect the 8086 and 68000 performance.

**Description of Benchmark Tests**

The benchmark tests used in this performance evaluation were selected for variety and are representative of applications including data processing, image processing and arithmetic processing. Detailed coding is shown in the appendix.

**Automated Parts Inspection**

The automated parts inspection program controls the interface to an image-dissector camera, and compares the gray shade signal from each of 16,384 points to a reference gray shade held in memory. The program controls the X-Y scan control to the camera by means of two 7-bit D-A converters and reads the resultant gray shade signal via a 12-bit A-D converter.

---

Reprinted with permission of Advanced Micro Devices
Figure 1  Relative Performance as a Function of Clock Frequency
Maximum frequencies are shown for available speed selections. Dotted lines indicate planned extensions.
Bubble Sort

The bubble sort is a well-known algorithm for sorting data elements into one sequence (in this case, numerically ascending order). The benchmark assumes that a one-dimensional array of ten elements is to be sorted and that the elements are initially in numerically descending order.

Array(0)

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>750</td>
</tr>
<tr>
<td>1</td>
<td>700</td>
</tr>
<tr>
<td>2</td>
<td>650</td>
</tr>
<tr>
<td>3</td>
<td>600</td>
</tr>
<tr>
<td>4</td>
<td>550</td>
</tr>
<tr>
<td>5</td>
<td>500</td>
</tr>
<tr>
<td>6</td>
<td>450</td>
</tr>
<tr>
<td>7</td>
<td>400</td>
</tr>
<tr>
<td>8</td>
<td>350</td>
</tr>
<tr>
<td>9</td>
<td>300</td>
</tr>
</tbody>
</table>

Array(0)

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>300</td>
</tr>
<tr>
<td>1</td>
<td>350</td>
</tr>
<tr>
<td>2</td>
<td>400</td>
</tr>
<tr>
<td>3</td>
<td>450</td>
</tr>
<tr>
<td>4</td>
<td>500</td>
</tr>
<tr>
<td>5</td>
<td>550</td>
</tr>
<tr>
<td>6</td>
<td>600</td>
</tr>
<tr>
<td>7</td>
<td>650</td>
</tr>
<tr>
<td>8</td>
<td>700</td>
</tr>
<tr>
<td>9</td>
<td>750</td>
</tr>
</tbody>
</table>

X:Y Transformation

The XY transformation scales a selected graphic window containing 16-bit unsigned integer XY pairs. Each X data is offset by XO and multiplied by a fractional scale factor L2/L1. Each Y data is offset by YO and multiplied by the same scale factor. The benchmark assumes the selected window contains 16,384 XY pairs.

Computer Graphics XY Transformation

This flowchart was originally presented by Intel

Reentrant Procedure

This benchmark demonstrates the ability of the processor to handle reentrant procedures and parameter passing between procedures. The input parameters are passed (by value) to the procedures. Prior to the call, the first parameter is in one of the general registers while the second and third parameters are stored in memory locations PARAM2 and PARAM3, respectively.

Upon entry, the procedure preserves the state of the processor, and it is assumed that the procedure uses eight of the general-purpose registers. Next, the procedure allocates the storage for three local variables (LOCAL1, LOCAL2, LOCAL3). The procedure then adds the three passed parameters and stores the result in the first local variable. Upon exit from the procedure, the state of the processor is restored.

Table 1 shows execution times for each benchmark on each microprocessor without and with one Wait State. Execution times are then inverted to indicate performance (not time), and normalized with respect to the slowest device, the 5MHz iAPX 86/10 (i.e. the original 8086). As can be seen from the detail data in the appendix, the Z8001 and Z8002 are so similar in performance that they can be grouped together.

Figure 1 shows the average performance data graphically.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Z8000B (8MHz)</th>
<th>Z8000A (8MHz)</th>
<th>Z8000 (4MHz)</th>
<th>68000-10 (10MHz)</th>
<th>68000-8 (8MHz)</th>
<th>iAPX 86/10 (10MHz)</th>
<th>iAPX 86/10 (8MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0W</td>
<td>1W</td>
<td>0W</td>
<td>1W</td>
<td>0W</td>
<td>1W</td>
<td>0W</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Absolute Performance</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Auto Parts Inspection</td>
<td>478</td>
<td>508</td>
<td>637</td>
<td>677</td>
<td>956</td>
<td>1016</td>
<td>470</td>
</tr>
<tr>
<td>Block Translation</td>
<td>388</td>
<td>456</td>
<td>517</td>
<td>607</td>
<td>776</td>
<td>912</td>
<td>757</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>539</td>
<td>646</td>
<td>718</td>
<td>861</td>
<td>1078</td>
<td>1292</td>
<td>507</td>
</tr>
<tr>
<td>XY Transformation</td>
<td>793</td>
<td>827</td>
<td>1057</td>
<td>1103</td>
<td>1585</td>
<td>1655</td>
<td>777</td>
</tr>
<tr>
<td>Reentrant Procedure</td>
<td>256</td>
<td>325</td>
<td>34</td>
<td>43</td>
<td>51</td>
<td>65</td>
<td>25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Performance Relative To iAPX 86/10 @ 5MHz</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Auto Parts Inspection</td>
<td>2.8</td>
<td>2.63</td>
<td>2.1</td>
<td>1.97</td>
<td>1.4</td>
<td>1.31</td>
<td>2.84</td>
<td>2.68</td>
<td>2.27</td>
<td>2.14</td>
</tr>
<tr>
<td>Block Translation</td>
<td>3.84</td>
<td>3.26</td>
<td>2.88</td>
<td>2.45</td>
<td>1.92</td>
<td>1.63</td>
<td>1.96</td>
<td>1.62</td>
<td>1.57</td>
<td>1.3</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>3.38</td>
<td>2.82</td>
<td>2.54</td>
<td>2.12</td>
<td>1.69</td>
<td>1.41</td>
<td>3.6</td>
<td>2.97</td>
<td>2.87</td>
<td>2.38</td>
</tr>
<tr>
<td>XY Transformation</td>
<td>2.82</td>
<td>2.71</td>
<td>2.12</td>
<td>2.03</td>
<td>1.41</td>
<td>1.35</td>
<td>2.88</td>
<td>2.79</td>
<td>2.3</td>
<td>2.23</td>
</tr>
<tr>
<td>Reentrant Procedure</td>
<td>2.42</td>
<td>1.9</td>
<td>1.82</td>
<td>1.44</td>
<td>1.21</td>
<td>0.95</td>
<td>2.48</td>
<td>2.00</td>
<td>1.93</td>
<td>1.59</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Average Relative Performance</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.05</td>
<td>2.66</td>
<td>2.28</td>
<td>1.99</td>
<td>1.53</td>
<td>1.34</td>
<td>2.75</td>
<td>2.4</td>
<td>2.19</td>
<td>1.93</td>
</tr>
<tr>
<td></td>
<td>2.00</td>
<td>1.84</td>
<td>1.60</td>
<td>1.48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0W = No Wait State, 1W = One Wait State per memory access.

### Table 1

#### Memory Access Time

The benchmark data compares the performance of the three microprocessors at nominal clock rates without regard to the memory access time required to achieve the performance.

Memory speed is, however, an important systems consideration since it has a strong impact on memory cost and the design of the supporting circuitry. In most systems memory cost far exceeds the cost of the CPU. It is therefore more useful to treat the CPU clock frequency as a variable and plot performance as a function of memory access time requirement. For each CPU, the memory access time requirement can be relaxed by using a higher speed version of the CPU, by lowering the actual clock frequency, or by adding Wait States.

Data sheets for the various microprocessors indicate the relationship between memory access time and clock period. Every Wait State adds another clock period to the memory access time.

\[
T_{AC} = (K + W)T - D
\]

- \(T_{AC}\) = memory access time required (at CPU pins)
- \(K\) = clock cycles/access (K=3 for the 8086; K=2.5 for the Z8000 and 68000)
- \(W\) = number of Wait States inserted (usually 0 or 1)
- \(T\) = actual clock period in ns
- \(D\) = sum of time for CPU delays, set-up times, etc. This is a constant for a given part type and speed selection. See Table for value.

### Table 2 Memory Access Times Required

<table>
<thead>
<tr>
<th>Device and Speed Selection</th>
<th>(f_{max})</th>
<th>D</th>
<th>T(<em>{AC}) in nanoseconds for various actual T (W = 0, T &lt; (\frac{1}{f</em>{max}}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z8001</td>
<td>4MHz</td>
<td>150ns</td>
<td>250ns</td>
</tr>
<tr>
<td>Z8002</td>
<td>4MHz</td>
<td>150ns</td>
<td>475</td>
</tr>
<tr>
<td>Z8001A</td>
<td>6MHz</td>
<td>95</td>
<td>530</td>
</tr>
<tr>
<td>Z8001B</td>
<td>8MHz</td>
<td>75</td>
<td>550</td>
</tr>
<tr>
<td>Z8002B</td>
<td>8MHz</td>
<td>75</td>
<td>550</td>
</tr>
<tr>
<td>68000-4</td>
<td>4MHz</td>
<td>120</td>
<td>505</td>
</tr>
<tr>
<td>68000-8</td>
<td>8MHz</td>
<td>90</td>
<td>535</td>
</tr>
<tr>
<td>68000-10</td>
<td>10MHz</td>
<td>80</td>
<td>545</td>
</tr>
<tr>
<td>8086-5</td>
<td>5MHz</td>
<td>140</td>
<td>610</td>
</tr>
<tr>
<td>8086-8</td>
<td>8MHz</td>
<td>80</td>
<td>670</td>
</tr>
<tr>
<td>8086-10</td>
<td>10MHz</td>
<td>60</td>
<td>690</td>
</tr>
</tbody>
</table>

- Device and Speed Selection:
  - Z8001: 4MHz, 150ns
  - Z8002: 4MHz, 150ns
  - Z8001A: 6MHz, 95
  - Z8001B: 8MHz, 75
  - Z8002B: 8MHz, 75
  - 68000-4: 4MHz, 120
  - 68000-8: 8MHz, 90
  - 68000-10: 10MHz, 80
  - 8086-5: 5MHz, 140
  - 8086-8: 8MHz, 80
  - 8086-10: 10MHz, 60

- \(T_{AC}\) in nanoseconds for various actual T (W = 0, T < \(\frac{1}{f_{max}}\))
  - 250ns (4MHz)
  - 167ns (6MHz)
  - 125ns (8MHz)
  - 100ns (10MHz)
The relative performances computed previously are obviously directly proportional to the clock frequency used. That is, for a given device selection, the relative performance is inversely proportional to \( T \), the actual clock period. The memory access time requirement is also related to the clock period.

\[
T_{AC} + D = (K + W) T = K_1 T
\]

and,

\[
RP = \frac{K_2}{T}
\]

Therefore,

\[
RP = \frac{K_1 K_2}{T_{AC} + D}
\]

and Relative Performance can be plotted against memory access time required, with the clock frequency being allowed to vary as required, down from the maximum for the part selection. As the clock frequency is reduced, a point is reached where equal performance can be achieved by raising the clock frequency back up and inserting a Wait State. This results in the same performance but a lower memory access time requirement, so it is logical to do so.

Table 3 contains computed data of memory access time requirements as a function of relative performance for each device selection with 0 and 1 Wait States. Figure 2 plots this data and shows the point at which the Wait State can be inserted without reducing performance.

### Relative Performance

<table>
<thead>
<tr>
<th>Performance</th>
<th>3.5</th>
<th>3.0</th>
<th>2.5</th>
<th>2.0</th>
<th>1.5</th>
<th>1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>Z8000B</td>
<td>68000-10</td>
<td>add 1 Wait State</td>
<td>68000-8</td>
<td>iAPX 86/10 (10MHz)</td>
<td>add 1 Wait State</td>
</tr>
</tbody>
</table>

**Fig. 2 Relative Performance as a Function of Memory Access Time**

Wait States are inserted when they reduce access time requirements without affecting performance (clock frequency is raised).
<table>
<thead>
<tr>
<th>Relative Performance</th>
<th>Z8000B (f ≤ 8MHz) W=0</th>
<th>W=1</th>
<th>Z8000A (f ≤ 6MHz) W=0</th>
<th>W=1</th>
<th>Z8000 (f ≤ 4MHz) W=0</th>
<th>W=1</th>
<th>68000-10 (f ≤ 10MHz) W=0</th>
<th>W=1</th>
<th>68000-8 (f ≤ 8MHz) W=0</th>
<th>W=1</th>
<th>iAPX 86/10 (f ≤ 10MHz) W=0</th>
<th>W=1</th>
<th>iAPX 86/10 (f ≤ 8MHz) W=0</th>
<th>W=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4</td>
<td>2.54</td>
<td></td>
<td>2.66</td>
<td></td>
<td>2.77</td>
<td></td>
<td>2.32</td>
<td></td>
<td>2.34</td>
<td></td>
<td>2.38</td>
<td></td>
<td>2.41</td>
<td></td>
</tr>
<tr>
<td>3.0</td>
<td>243</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.9</td>
<td>254</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.8</td>
<td>266</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.7</td>
<td>279</td>
<td></td>
<td>175</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.6</td>
<td>292</td>
<td>373</td>
<td>184</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.5</td>
<td>307</td>
<td>391</td>
<td>195</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td>323</td>
<td>410</td>
<td>206</td>
<td>270</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td>340</td>
<td>432</td>
<td>219</td>
<td>285</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td>359</td>
<td>455</td>
<td>335</td>
<td></td>
<td>233</td>
<td>302</td>
<td>221</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td>380</td>
<td>480</td>
<td>356</td>
<td></td>
<td>247</td>
<td>320</td>
<td>235</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.0</td>
<td>402</td>
<td>508</td>
<td>378</td>
<td>486</td>
<td>264</td>
<td>340</td>
<td>252</td>
<td>240</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.9</td>
<td>427</td>
<td>538</td>
<td>403</td>
<td>517</td>
<td>282</td>
<td>362</td>
<td>270</td>
<td>354</td>
<td>256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.8</td>
<td>455</td>
<td>572</td>
<td>431</td>
<td>551</td>
<td>302</td>
<td>387</td>
<td>290</td>
<td>379</td>
<td>273</td>
<td>349</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.7</td>
<td>487</td>
<td>610</td>
<td>462</td>
<td>589</td>
<td>324</td>
<td>414</td>
<td>312</td>
<td>406</td>
<td>293</td>
<td>373</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.6</td>
<td>522</td>
<td>653</td>
<td>496</td>
<td>631</td>
<td>350</td>
<td>445</td>
<td>337</td>
<td>437</td>
<td>315</td>
<td>400</td>
<td>295</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.5</td>
<td>561</td>
<td>702</td>
<td>536</td>
<td>680</td>
<td>378</td>
<td>480</td>
<td>366</td>
<td>472</td>
<td>340</td>
<td>431</td>
<td>320</td>
<td>413</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.4</td>
<td>607</td>
<td>757</td>
<td>581</td>
<td>735</td>
<td>411</td>
<td>520</td>
<td>398</td>
<td>512</td>
<td>369</td>
<td>466</td>
<td>349</td>
<td>449</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.3</td>
<td>659</td>
<td>821</td>
<td>633</td>
<td>799</td>
<td>449</td>
<td>566</td>
<td>436</td>
<td>559</td>
<td>402</td>
<td>506</td>
<td>382</td>
<td>489</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td>721</td>
<td>896</td>
<td>694</td>
<td>873</td>
<td>493</td>
<td>620</td>
<td>479</td>
<td>613</td>
<td>440</td>
<td>553</td>
<td>420</td>
<td>537</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>793</td>
<td>984</td>
<td>765</td>
<td>961</td>
<td>545</td>
<td>684</td>
<td>531</td>
<td>677</td>
<td>485</td>
<td>609</td>
<td>465</td>
<td>593</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>880</td>
<td>1090</td>
<td>851</td>
<td>1067</td>
<td>608</td>
<td>760</td>
<td>593</td>
<td>753</td>
<td>540</td>
<td>676</td>
<td>520</td>
<td>660</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

W=0 = No Wait State, W=1 = One Wait State per memory access

Table 3: Required Memory Access Time to Achieve a Given Relative Performance (in nanoseconds)

What This Benchmark Does And Doesn't Tell You

Benchmarks are popular simplifications to compare the performance of different microprocessors. Like all other simplifications, benchmarks must be used with care.

At best they accurately compare the performance of different microprocessors in a limited set of applications, which may or may not be representative of the applications that the user needs.

At worst they are distorted by a manufacturer who wants to "prove" that his device is the best. By choosing examples that favor a particular microprocessor or — more deviously — by writing inefficient code for the competitor’s device, any manufacturer can "prove" that his product is superior to the competition’s.

Moreover, benchmarks describe only one aspect of the microprocessor: speed (or throughput). Other important technical considerations are:

- Code efficiency
- Ease of programming
- Ease of interfacing to memory and I/O
- Availability of powerful peripheral devices
- Availability of hardware and software support

Finally there are good business reasons for favoring a particular microprocessor:

- Price, availability and multiple sourcing
- Vendor reputation and quality of field application support
- Device reliability and quality level.

Benchmarks tell nothing about these important aspects.

In spite of these limitations, benchmarks are an important tool for adding quantitative data to the complicated task of selecting the right microprocessor.

The soon-to-be-announced 8MHz Z8000B is 11% faster than the soon-to-be-announced 10MHz 68000-10, and the Z8000B achieves this superior performance even with substantially slower memories.

The 6MHz Z8000A is 4% faster than the 8MHz 68000-8, and the Z8000A can tolerate memory access times 100ns longer than required by the 68000-8. The iAPX 86, even in its fastest 10MHz version is no contender.

The Z8000 is better.
APPENDIX
A. Automated Parts Inspection

Z8002 # of Clock Cycles

LD R12, PERCENT 7 + 2W
LD R8, ↑GRAYTAB 11 + 3W
LD R0, 16383 7 + 2W
LD R10, SIGNAL 7 + 2W
LD R11, XYSCAN 7 + 2W
LD R13, REJECT 7 + 2W
LOOP OUT R11, R0 7 + 2W
IN R4, R10 Z = R4 (Read Signal) 10 + W
LD R3, R8 ↑ Z0 = R3 (Read Reference) 7 + 2W
INC R8, 2 Inc Reference Pointer 3 + W
LD R1, R3 R1 = Z0 3 + W
MUL RR2, R12 R3 = Z0 * PERCENT 70 + W
DIV RR2, #100 R3 = Z0 * PERCENT/100 95 + 2W
SUB R4, R1 R4 = Z0 - Z0 4 + W
JR GE BYPASS R4 = 0 6 + W
NEG R4 R4 < 0 → R4 = | Z-Z0 | 7 + W
BYPASS CP R4, R3 | Z-Z0 | - Z0 * PERCENT/100 4 + W
JR LE ENDTEST | Z-Z0 | - Z0 * PERCENT/100 2 + W
OUT R13, R4 Reject Signal 10 + W
ENDTEST DJNZ R0, LOOP Process Next Point 4 + W

CONSTANT PERCENT =
CONSTANT SIGNAL =
CONSTANT XYSCAN =
CONSTANT REJECT =
GRAYTAB WORD (16384)

On average, of 16384 times through Loop we assume that 8192 times
Z-Z0 > 0
8192 times Z-Z0 < 0 we execute NEG R4
1638 times (10% of the cases) we reject the point, ie we execute
OUT R13, R4
Total Clocks 6(7 + 2W) + 8192 (229 + 14W) + 8192 (236 + 15W) + 1638(10 + W) = 16422 + 1650W + 8192 (465 + 29W) = 3,825,702 + 239,218W

Z8001 # of Clock Cycles

LD R12, PERCENT 7 + 2W
LDL R8, ↑GRAYTAB 11 + 3W
LD R0, 16383 7 + 2W
LD R10, SIGNAL 7 + 2W
LD R11, XYSCAN 7 + 2W
LD R13, REJECT 7 + 2W
LOOP OUT R11, R0 10 + W
IN R4, R0 10 + W
LD R3, R8 ↑ 7 + 2W
INC R9, 2 3 + W
LD R1, R3 3 + W
MUL RR2, R12 70 + W
DIV RR2, #100 95 + 2W
SUB R4, R1 4 + 2W
JR GE BYPASS 6 + W
NEG R4 7 + W

8192 times Z-Z0 > 0
8192 times Z-Z0 < 0 we execute NEG R4
1638 times (10% of the cases) we reject the point, ie we execute
OUT R13, R4
Total Clocks 6(7 + 2W) + 8192 (229 + 14W) + 8192 (236 + 15W) + 1638(10 + W) = 16422 + 1650W + 8192 (465 + 29W) = 3,825,702 + 239,218W

Z8001 (Continued) # of Clock Cycles

BYPASS CP R4, R3 4 + W
JR LE ENDTEST 6 + W
OUT R13, R4 10 + W
ENDTEST DJNZ R0, LOOP 11 + W

Total clocks. 3,825,702 + 239,219 W
Notice that there is practically no performance deterioration due to segmentation

68000 # of Clock Cycles

MOVEW D0, #16383 9 + 2W
MOVEW D6, #PERCENT 8 + 2W
MOVEW D6, #PERCENT 8 + 2W
MOVEL A3, #GRAYTAB 12 + 3W
MOVEW A5, #XYSCAN 8 + 2W
MOVEW A6, #REJECT 8 + 2W
MOVEW A4, #SIGNAL 8 + 2W
MOVEW (A5), D0 14 + 2W

BGE BYPASS 9 + 2W
NEGW D4 8 + 2W
NEGW D4 4 + W
BYPASS CMPW D4, D3 144 + 2W
BLE ENDTEST 14 + 2W
NEGW D4 8 + 2W

ENDTEST DBF D0, LOOP 14 + 2W

Total clocks. 52 + 13W + 8192 (285 + 11W) + 8192 (287 + 18N) + 1638 (8-2W+2W)= 52 + 13W + 8192 (572 + 35W) + 1638 (6 +2W)= 4,695,576 + 290 009W

iAPX 86/10 # of Clock Cycles

XOR CX, CX 3
MOV SI, OFFSET(GDATA) 4 + W
CLOD 2
AGAIN MOV AX, CX 2
OUT DTOA, AX 10 + W
LODS GDATA 12 + W

4-15
<table>
<thead>
<tr>
<th>iAPX 86/10 (Continued)</th>
<th># of Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV BX, AX</td>
<td>STORE ZO IN BX</td>
</tr>
<tr>
<td>MUL PERCNT</td>
<td>Z0 PERCNT</td>
</tr>
<tr>
<td>OUT CONVRT,AX</td>
<td>START A/D CONVERTER</td>
</tr>
<tr>
<td>DIV HUNDRED</td>
<td>.Z0*PERCNT/100</td>
</tr>
<tr>
<td>MOV DX, AX</td>
<td>.DX=TOLER 2</td>
</tr>
<tr>
<td>IN AX, ATOD</td>
<td>.INPUT Z FROM A/D</td>
</tr>
<tr>
<td>SUB AX, BX</td>
<td>.DELTA=-Z-Z0 3</td>
</tr>
<tr>
<td>JA CMPARE</td>
<td>.JUMP IF PRINT</td>
</tr>
<tr>
<td>NEG AX</td>
<td>.DELTA-=DELTA 3</td>
</tr>
<tr>
<td>CMPARE: CMP AX, DX</td>
<td>.DELTA&lt;=DELTA 3</td>
</tr>
<tr>
<td>JBE INCCX</td>
<td>.JUMP IF YES 4/16</td>
</tr>
<tr>
<td>OUT REJECT, AX</td>
<td>.REJECT PART 10+</td>
</tr>
<tr>
<td>JMP SHORT (NEXT)</td>
<td>.INC X = 2</td>
</tr>
<tr>
<td>INCCX INC CX</td>
<td>.INC X = Y 2</td>
</tr>
<tr>
<td>CMP CX, 4000H</td>
<td>.DONE? 4 + W</td>
</tr>
<tr>
<td>JNE AGAIN</td>
<td>.NO, PROCESS 4/16</td>
</tr>
<tr>
<td>NEXT</td>
<td>NEXT POINT</td>
</tr>
<tr>
<td>HUNDRED DW 100</td>
<td>HUNDRED DW 100</td>
</tr>
</tbody>
</table>

Total number of clock cycles 6,680,000 + 400W.

Block Translate — Destructive
(Special feature for Z8000) # of Clock Cycles

<table>
<thead>
<tr>
<th>Z8002 (Continued)</th>
<th># of Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R3, EBCBUF</td>
<td>Address of EBCDIC</td>
</tr>
<tr>
<td>LD R2, EBCEOT</td>
<td>EOT in EBCDIC</td>
</tr>
<tr>
<td>LD R0, COUNT</td>
<td>COUNT = COUNT</td>
</tr>
<tr>
<td>LD R1, R0</td>
<td>COUNT = COUNT + 3</td>
</tr>
<tr>
<td>CMP IRB R2, R3, R0, EQ</td>
<td>Address of Translation Table</td>
</tr>
<tr>
<td>SUB R1, R0</td>
<td>R1 = R0 = R0</td>
</tr>
<tr>
<td>CMP R3, EBCBUF</td>
<td>Address of EBCDIC</td>
</tr>
<tr>
<td>CMP R5, TRTAB</td>
<td>Address of EBCDIC</td>
</tr>
<tr>
<td>TRIRB R3, R5, R1</td>
<td>Table 7 + 2W</td>
</tr>
<tr>
<td>LDB R3, ASCEOT</td>
<td>Write ASCEOT</td>
</tr>
</tbody>
</table>

Total clocks 3111 + 547W
This is the worst possible case since the scanning of the string is actually done only for characters (until the encounter of EOT).

Z8001

<table>
<thead>
<tr>
<th># of Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRTAB EBCBUF</td>
</tr>
<tr>
<td>CONSTANT EBCEOT = 3</td>
</tr>
<tr>
<td>CONSTANT COUNT = 132</td>
</tr>
<tr>
<td>CONSTANT ASCEOT = 04</td>
</tr>
<tr>
<td>LDL RR2, EBCEOT</td>
</tr>
<tr>
<td>LDL R4, EBCEOT</td>
</tr>
<tr>
<td>LDL R0, COUNT</td>
</tr>
<tr>
<td>LDL R1, R0</td>
</tr>
<tr>
<td>CPIR R4, R2, R0, EQ</td>
</tr>
<tr>
<td>SUB R1, R0</td>
</tr>
<tr>
<td>LDL RR2, EBCEOT</td>
</tr>
<tr>
<td>LDL RR6, TRTAB</td>
</tr>
<tr>
<td>TRIRB RR2, RR6, R1</td>
</tr>
<tr>
<td>LDB RR2, ASCEOT</td>
</tr>
</tbody>
</table>

Total clocks 3123 + 550W

68000

<table>
<thead>
<tr>
<th># of Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVEB D2, #EOT</td>
</tr>
<tr>
<td>MOVEW D0, #COUNT</td>
</tr>
<tr>
<td>MOVEB A3, EBCEOT</td>
</tr>
<tr>
<td>MOVEV A5, #TRTAB</td>
</tr>
<tr>
<td>LOOP MOVEB D1, (A3)</td>
</tr>
<tr>
<td>MOVEB (A3), A5, (0, D1)</td>
</tr>
<tr>
<td>CMPB D2, (A3)+</td>
</tr>
</tbody>
</table>

Total clocks 1880 + 404W

B. Block Translate Benchmark — Destructive

<table>
<thead>
<tr>
<th>Z8002</th>
<th># of Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRTAB</td>
<td>ICICEBD-ASCII Translation Table</td>
</tr>
<tr>
<td>EBCBUF</td>
<td>CONSTANT EBCEOT = 03</td>
</tr>
<tr>
<td></td>
<td>CONSTANT COUNT = 132</td>
</tr>
<tr>
<td></td>
<td>CONSTANT ASCEOT = 04</td>
</tr>
</tbody>
</table>

Total Clocks 1880 + 404W

<table>
<thead>
<tr>
<th>68000</th>
<th># of Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVEB D2, #EOT</td>
<td></td>
</tr>
<tr>
<td>MOVEW D0, #COUNT</td>
<td></td>
</tr>
<tr>
<td>MOVEB A3, EBCEOT</td>
<td></td>
</tr>
<tr>
<td>MOVEV A5, #TRTAB</td>
<td></td>
</tr>
<tr>
<td>LOOP MOVEB D1, (A3)</td>
<td></td>
</tr>
<tr>
<td>MOVEB (A3), A5, (0, D1)</td>
<td></td>
</tr>
<tr>
<td>CMPB D2, (A3)+</td>
<td></td>
</tr>
</tbody>
</table>

Total clocks 1880 + 404W
68000 (Continued) \( \# \) of Clock Cycles

\[ \text{Total clocks:} \quad 48 + 11W + 132(57 + 12W) - (4 + W) = 44 + 10W + 7524 + 1584W = 7568 + 1594W \]

**IA68 86/10** \( \# \) of Clock Cycles

| MOV BX,OFFSET(TABLE) | .INIT TRANSLATION PTR 4 |
| MOV SI,OFFSET(EBCBUF) | .INIT EBCDIC BUF PTR 4 |
| MOV DI,OFFSET(ASCBUF) | .INIT ASCII BUF PTR 4 |
| MOV CX,COUNT | .INIT CNT 14 + W |
| STOS ASCBUF | .STORE IN ASCII BUFFR 11 + W |
| CMP AL,EOT | .CHAR=EOT? 4 |
| LOOPNE NEXT | .LOOP IF NE OR CX<0 5/19 + W |

**68000** \( \# \) of Clock Cycles

| BSORT LD R4,ADR | .Load Starting Address 9 + 3W |
| LD R5,COUNT | .Load Word Count 9 + 3W |
| DEC R5 | .Set Number of Compares 4 + W |
| 10 | .Clear Exchange Flag 4 + W |
| * | .Fetch 2 words in R0,R1 11 + 2W |
| * | .Out of Order? 4 + W |
| * | .JE DECNT No-Continue 6 + W |
| * | .EX R0,R1 Yes-Swap them 6 + W |
| * | .SETRL6,0; Store Back 11 + 2W |
| * | .DECCNT INC R2,2; Point to Next Pair 4 + W |
| * | .DEC R3 Decr, Word Count 4 + W |
| * | .JRGTCOMP .Done? 6 + W |
| 10 | .Exchange Flag = ? 4 + W |
| * | .JRNZINIT .Yes-Start Next Pass 6 + W |

**Z8002** \( \# \) of Clock Cycles

| COMP: LDLR0,R12 | .Exchange=TRUE 11 + 2W |
| CP R0,R1 | 4 + W |
| JR LE DECNT | 6 + W |
| EX R0,R1 | 6 + W |
| LDL RR21,RRO | 11 + 2W |
| SETBL6,0 | 4 + W |
| DECCNT INC R3,2 | 4 + W |
| DEC R4 | 4 + W |
| JRGTCOMP | 4 + W |
| BITBL6,0 | 4 + W |
| 10 | .JRNZINIT 6 + W |

**BSORT** MOVEAL A1,400 | .Start Address->A1 12 + 3W |
| MOV EW D3,404 | .Count->D3 12 + 3W |
| SUBQ D3,#1 | 4 + W |
| CLR B D1 | .Exchange Flag = 0 4 + W |
| MOV D0,D3 | .Copy Count into D0 4 + W |
| * | .Fetch word 8 + 2W |
| * | .Next word greater? 8 + 2W |
| * | .BLS S DECCNT .Yes, Continue 8/10 + W |
| MOV EW (A0)(-2),D0 | .No, Exchange these 17 + 4W |
| MOV EW (A0),D2 | .two words 9 + 2W |
| TAE D1 | .Exchange Flag=1 4 + 3W |
| * | .DECCNT DB E D0,COMP .Done? 10 + 2W/14 + 3W |
| 10 | .BPL S INIT 8/10 + W |

**Z8001** \( \# \) of Clock Cycles

| BSORT LDL RR12,ADR | .Load Starting Address 15 + 4W/13 + 3W |
| LDL RR2,RR12 | .Exchange=TRUE 4 |
| LDL RR2,RRA | 11 + 2W |
| LDL RR2,RR4 | 4 + W |
| MOVEW D2,(AO)+ | 10 + 2W/14 + 3W |
| MOVEW 00,03 | 8/10 + W |
| CMP (AO),D2 | 4 + 3W |
| 10 | .JRNZINIT .Yes-Start Next Pass 6 + W |

**IA68 86/10** \( \# \) of Clock Cycles

| MOV BL,OFFH | .EXCHANGE=TRUE 4 |
| CMP BL,OFFH | 4 + W |
| JNE A4 | .NO, FINISHED 4/16 + W |
| XOR BL,BL | .EXCHANGE=FALSE 3 |
| MOV CX,COUNT | .CX=COUNT-1 14 + W |
| DEC CX | 2 |
| XOR SI,SI | .SI=0 3 |
| A2: MOV AX,ARRAY(SI) | .ARRAY(I) > 17 + W |
| CMP AX,ARRAY(SI+2) | 18 + W |
| JLE A3 | .NO 4/16 + W |
| XCHG ARRAY(SHZ),AX | .EXCHANGE ELEMENTS 6 + W |
| ARRAY(SI),AX | 18 + W |
### 68000 (Continued)  

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycles</th>
<th># of Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV E4,X0</td>
<td>INIT X0</td>
<td>12 + 3W</td>
</tr>
<tr>
<td>MOV E5,Y0</td>
<td>INIT Y0</td>
<td>12 + 3W</td>
</tr>
<tr>
<td>MOV E6,L2</td>
<td>INIT L2</td>
<td>12 + 3W</td>
</tr>
<tr>
<td>MOV E7,L1</td>
<td>INIT L1</td>
<td>12 + 3W</td>
</tr>
</tbody>
</table>

#### XYSCAL:  

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycles</th>
<th># of Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV E4,D1(A3)</td>
<td>GET X</td>
<td>8 + 2W</td>
</tr>
<tr>
<td>SUBW D1,D4</td>
<td>X=X0</td>
<td>4 + W</td>
</tr>
<tr>
<td>MULU D1,D6</td>
<td>(X-X0)*L2</td>
<td>70 + W</td>
</tr>
<tr>
<td>DIVU D1,D7</td>
<td>(X-X0)*L2/L1</td>
<td>140 + W</td>
</tr>
<tr>
<td>MOV E4(A3)+,D1</td>
<td>STORE &amp; INC POINTER</td>
<td>8 + 2W</td>
</tr>
</tbody>
</table>

Total clocks: 64 + 16W + 16386 (474 + 17W) = 7,766,016 + 278,544W

### 86/10 (Continued)  

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycles</th>
<th># of Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV CX,COUNT</td>
<td>INIT COUNT</td>
<td>14 + W</td>
</tr>
<tr>
<td>MOV SI,OFFSET(ARRAY)</td>
<td>INIT ARRAY POINTER</td>
<td>4</td>
</tr>
<tr>
<td>MOV DI,SI</td>
<td>INIT ARRAY POINTER</td>
<td>2</td>
</tr>
<tr>
<td>CLD</td>
<td>DF=FORWARD</td>
<td>2</td>
</tr>
</tbody>
</table>

#### XYSCAL:  

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycles</th>
<th># of Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lods ARRAY</td>
<td>GET X ELEMENT</td>
<td>12 + W</td>
</tr>
<tr>
<td>SUB AX,X0</td>
<td>X=X0</td>
<td>15 + W</td>
</tr>
<tr>
<td>MUL L2</td>
<td>(X-X0)*L2</td>
<td>130 + W</td>
</tr>
<tr>
<td>DIVL1</td>
<td>(X-X0)*L2/L1</td>
<td>161 + W</td>
</tr>
<tr>
<td>STOS ARRAY</td>
<td>STORE ELEMENT</td>
<td>11 + W</td>
</tr>
<tr>
<td>LODS ARRAY</td>
<td>GET Y ELEMENT</td>
<td>12 + Y</td>
</tr>
<tr>
<td>SUB AX,Y0</td>
<td>Y=Y0</td>
<td>15 + W</td>
</tr>
<tr>
<td>MUL L2</td>
<td>(Y-Y0)*L2</td>
<td>130 + W</td>
</tr>
<tr>
<td>DIVL1</td>
<td>(Y-Y0)*L2/L1</td>
<td>161 + W</td>
</tr>
<tr>
<td>STOS ARRAY</td>
<td>STORE ELEMENT</td>
<td>11 + W</td>
</tr>
<tr>
<td>LOOP XYSCAL</td>
<td>.DEC CX &amp; LOOP IF</td>
<td>5/17 + W</td>
</tr>
</tbody>
</table>

Total number of clock cycles = 11,200,000 + 320,000W

### E. Reentrant Procedure  

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycles</th>
<th># of Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH R15,R8</td>
<td>R8=PARAM1</td>
<td>9 + 2W</td>
</tr>
<tr>
<td>PUSH R15,PARAM2</td>
<td>PUSH PARAM2</td>
<td>13 + 4W</td>
</tr>
<tr>
<td>PUSH R15,PARAM3</td>
<td>PUSH PARAM3</td>
<td>13 + 4W</td>
</tr>
<tr>
<td>CALR PROC1</td>
<td></td>
<td>10 + W</td>
</tr>
<tr>
<td>INC R15,6</td>
<td>Remove PARAM1-3 from the Stack</td>
<td>4 + W</td>
</tr>
</tbody>
</table>

#### PROC1  

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Cycles</th>
<th># of Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH R15,R14</td>
<td>Save R14</td>
<td>9 + 2W</td>
</tr>
<tr>
<td>LD R14,R15</td>
<td>Initialize R14</td>
<td>3 + W</td>
</tr>
<tr>
<td>SUB R15,6+16</td>
<td>Set Up Local Storage</td>
<td>7 + 2W</td>
</tr>
</tbody>
</table>
Z8002 (Continued)  

<table>
<thead>
<tr>
<th># of Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDM R15,R0,8      : Save Registers R0-7</td>
</tr>
<tr>
<td>25 + 10W</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>PROCEDURE BODY</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>LD R0,(R14)       , Get PARAM1</td>
</tr>
<tr>
<td>10 + 3W</td>
</tr>
<tr>
<td>ADD R0,(R14)      , ADD PARAM2</td>
</tr>
<tr>
<td>10 + 3W</td>
</tr>
<tr>
<td>ADD R0,(R14)      , ADD PARAM3</td>
</tr>
<tr>
<td>10 + 3W</td>
</tr>
<tr>
<td>LD -2(R14),R0     , Store in LOCAL1</td>
</tr>
<tr>
<td>12 + 3W</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>PROCEDURE RETURN</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>LDM R0,R15       , Restore General Registers</td>
</tr>
<tr>
<td>35 + 10W</td>
</tr>
<tr>
<td>ADD R15,6+16      , Add to R14</td>
</tr>
<tr>
<td>7 + 2W</td>
</tr>
<tr>
<td>POP R14,R15      , Restore R14</td>
</tr>
<tr>
<td>18 + 2W</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Total clocks: 205 + 55W</td>
</tr>
</tbody>
</table>

Z8001  

<table>
<thead>
<tr>
<th># of Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH RR14,R8      , R8 = PARAM1</td>
</tr>
<tr>
<td>9 + 2W</td>
</tr>
<tr>
<td>PUSH RR14,PARAM2  , Push PARAM2</td>
</tr>
<tr>
<td>14 + 4W + 16 + 5W</td>
</tr>
<tr>
<td>PUSH RR14,PARAM3  , Push PARAM3</td>
</tr>
<tr>
<td>14 + 4W + 16 + 5W</td>
</tr>
<tr>
<td>CALR PROC1</td>
</tr>
<tr>
<td>15 + 3W</td>
</tr>
<tr>
<td>INC R15,6         , Remove PARAM1-3 from stack</td>
</tr>
<tr>
<td>4 + W</td>
</tr>
<tr>
<td>PROC1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>PUSHL,RR14,R12    , Save RR12</td>
</tr>
<tr>
<td>12 + 3W</td>
</tr>
<tr>
<td>LDL RR12,RR14     , Initialize RR12</td>
</tr>
<tr>
<td>5 + W</td>
</tr>
<tr>
<td>SUB R15,6+16      , Setup Local Storage</td>
</tr>
<tr>
<td>7 + 2W</td>
</tr>
<tr>
<td>LDM RR14,R0,8     , Save R0-7</td>
</tr>
<tr>
<td>35 + 10W</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>PROCEDURE BODY</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>LD R0,12(RR12)    , Get PARAM1</td>
</tr>
<tr>
<td>14 + 3W</td>
</tr>
<tr>
<td>LD R1,10(RR12)    , Add PARAM2</td>
</tr>
<tr>
<td>14 + 3W</td>
</tr>
<tr>
<td>ADD R0,R1         ,</td>
</tr>
<tr>
<td>4 + W</td>
</tr>
<tr>
<td>LD R1,(R12)       , Add PARAM3</td>
</tr>
<tr>
<td>14 + 3W</td>
</tr>
<tr>
<td>ADD R0,R1         ,</td>
</tr>
<tr>
<td>4 + W</td>
</tr>
<tr>
<td>LD -2(RR12),R0    , Store in LOCAL1</td>
</tr>
<tr>
<td>14 + 3W</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>PROCEDURE RETURN</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>LDM R0,8,RR14     , Restore R0-7</td>
</tr>
<tr>
<td>35 + 10W</td>
</tr>
<tr>
<td>ADD R15,6+16      , Add to RR12</td>
</tr>
<tr>
<td>7 + 2W</td>
</tr>
<tr>
<td>POPL RR12,RR14    , Restore RR12</td>
</tr>
<tr>
<td>12 + 3W</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Total clocks (Short segmentation): 243 + 60W</td>
</tr>
<tr>
<td>Total clocks (Long segmentation): 247 + 62W</td>
</tr>
</tbody>
</table>

68000  

<table>
<thead>
<tr>
<th># of Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOVEW -(SP),D0    , DO = PARAM1</td>
</tr>
<tr>
<td>9 + 2W</td>
</tr>
<tr>
<td>MOVEW -(SP),PARAM2 , Push PARAM2</td>
</tr>
<tr>
<td>17 + 3W</td>
</tr>
<tr>
<td>MOVEW -(SP),PARAM3 , Push PARAM3</td>
</tr>
<tr>
<td>17 + 3W</td>
</tr>
</tbody>
</table>

68000 (Continued)  

<table>
<thead>
<tr>
<th># of Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSR SUB</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>ADDQ SP,#6        , Remove PARAM1-3 from the stack</td>
</tr>
<tr>
<td>4 + W</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>SUB</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>LINK A6,#6        , A6 = Frame pointer</td>
</tr>
<tr>
<td>18 + 4W</td>
</tr>
<tr>
<td>MOVEMW OFFO,(SP)  , Save A3-0,D7-4 on</td>
</tr>
<tr>
<td>48 + 10W</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>PROCEDURE BODY</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>MOVEV D0,A6(+10)  , Get PARAM1</td>
</tr>
<tr>
<td>12 + 3W</td>
</tr>
<tr>
<td>ADDW D0,A6(+8)    , Add PARAM2</td>
</tr>
<tr>
<td>12 + 3W</td>
</tr>
<tr>
<td>ADD W D0,A6(+6)   , Add PARAM3</td>
</tr>
<tr>
<td>12 + 3W</td>
</tr>
<tr>
<td>MOVEV A6(-2),D0   , Store in LOCAL1</td>
</tr>
<tr>
<td>9 + 3W</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>PROCEDURE RETURN</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>MOVEMW (SP)+,OFFO , Restore A3-0,D7-4</td>
</tr>
<tr>
<td>44 + 11W</td>
</tr>
<tr>
<td>UNLK A6           , Restore A6</td>
</tr>
<tr>
<td>12 + 3W</td>
</tr>
<tr>
<td>RTS               , Restore A6</td>
</tr>
<tr>
<td>16 + 4W</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Total clocks: 250 + 58W</td>
</tr>
</tbody>
</table>

iAPX 86/10  

<table>
<thead>
<tr>
<th># of Clock Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH AX           , PUSH PARAM1</td>
</tr>
<tr>
<td>10 + W</td>
</tr>
<tr>
<td>PUSH PARAM2</td>
</tr>
<tr>
<td>22 + W</td>
</tr>
<tr>
<td>PUSH PARAM3</td>
</tr>
<tr>
<td>22 + W</td>
</tr>
<tr>
<td>CALL PROC1</td>
</tr>
<tr>
<td>19 + W</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Procedure ENTRY</td>
</tr>
<tr>
<td>PROC1</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>PUSH BP           , SAVE BP</td>
</tr>
<tr>
<td>10 + W</td>
</tr>
<tr>
<td>MOV BPSP          , INITIALIZE BP</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>SUB SP6           , SETUP LOCAL STORAGE</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>PUSH AX           , SAVE GENERAL</td>
</tr>
<tr>
<td>10 + W</td>
</tr>
<tr>
<td>PUSH BX           , REGISTERS</td>
</tr>
<tr>
<td>10 + W</td>
</tr>
<tr>
<td>PUSH CX</td>
</tr>
<tr>
<td>10 + W</td>
</tr>
<tr>
<td>PUSH DX</td>
</tr>
<tr>
<td>10 + W</td>
</tr>
<tr>
<td>PUSH SI</td>
</tr>
<tr>
<td>10 + W</td>
</tr>
<tr>
<td>PUSH DI</td>
</tr>
<tr>
<td>10 + W</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Procedure BODY</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>MOV AX,(BP+8)     , GET PARAM1</td>
</tr>
<tr>
<td>17 + W</td>
</tr>
<tr>
<td>ADD DX,(BP+6)     , ADD PARAM2</td>
</tr>
<tr>
<td>18 + W</td>
</tr>
<tr>
<td>ADD AX,(BP+4)     , ADD PARAM3</td>
</tr>
<tr>
<td>18 + W</td>
</tr>
<tr>
<td>MOV (BP-2),AX     , STORE IN LOCAL1</td>
</tr>
<tr>
<td>18 + W</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Procedure RETURN</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>POP DI            , RESTORE GENERAL</td>
</tr>
<tr>
<td>8 + W</td>
</tr>
<tr>
<td>POP SI            , REGISTERS</td>
</tr>
<tr>
<td>8 + W</td>
</tr>
<tr>
<td>POP DX</td>
</tr>
<tr>
<td>8 + W</td>
</tr>
<tr>
<td>POP CX</td>
</tr>
<tr>
<td>8 + W</td>
</tr>
<tr>
<td>POP BX</td>
</tr>
<tr>
<td>8 + W</td>
</tr>
<tr>
<td>POP AX</td>
</tr>
<tr>
<td>8 + W</td>
</tr>
<tr>
<td>MOV SPBP          , RESTORE SP</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>POP BP            , RESTORE BP</td>
</tr>
<tr>
<td>8 + W</td>
</tr>
<tr>
<td>RTS               , Restore A6</td>
</tr>
<tr>
<td>20 + W</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Total number of clock cycles = 310 + 35W</td>
</tr>
</tbody>
</table>

4-19
OPERATING SYSTEM SUPPORT—THE Z8000 WAY

All processor architectures are not created equal when it comes to providing designers with the tools they need for effective system resource management.

by Richard Mateosian

Operating systems are responsible for allocation, deallocation, and protection of processing and storage elements, external interfaces, programs, and program status. They manage communication and sharing, and define, facilitate, and enforce protocols, conventions, and policy. Several kinds of architectural support facilitate the operating system's task in a wide range of applications: restriction of central processing unit and memory use, memory mapping, sharing of programs and data, program relocation, stacks, context switching, input/output system and interrupts, distributed control, and support for conventions.

Operating system support is an important feature of Z8000 architecture. Special consideration was given to that function during design of the Z8000 central processing unit (CPU), the Z-BUS component interconnect, and their support chips. In this discussion, “operating system” will comprise the portion of the computer application—both hardware and software—that is devoted to managing hardware and software resources.

Richard Mateosian, Z8000 specialist at Zilog, Inc, 1315 Dell Ave, Campbell, CA 95008, is the author of Programming the Z8000 (Sybex 1980) and Inside BASIC Games (Sybex 1981). Formerly employed in the development of minicomputer based turnkey systems, he has a BS in mathematics from Rensselaer Polytechnic Institute and a PhD from the University of California at Berkeley.

*Z8000 and Z-BUS are registered trademarks of Zilog, Inc

Reprinted with permission of Computer Design, May 1982

Fig 1 Hardware block diagram of arcade game system. Essential elements include CPU, memory, input and display devices, and clock circuits.

To show how the Z8000 provides operating system support, an application of the hardware and software similar to that used in a popular arcade game will be described. Fig 1 shows the game's hardware configuration; the system elements are pieces of hardware including CPU, memory, realtime clock, input and display units, and integrated circuits for interface to the CPU. Arrows represent electrical connections through which data and control signals are passed among the elements. Configuration of the hardware elements alone, however, provides little insight into the game's operation.

In the game's software architecture (Fig 2), system elements are pieces of software "in action" on the data defining the state of play at any time. Connecting
Restriction of CPU access

The operating system must allocate the CPU to a process while protecting itself and other processes. In other words, the operating system must be able to turn the CPU over to a process that will not perform potentially destructive actions. To this end, the Z8000 incorporates a system/normal (S/N) bit in its flag/control word (FCW) register, which corresponds to the program status word (PSW) in other machines. (See Fig 4.) The S/N bit determines whether the CPU executes in system or normal mode. In normal mode, the portion of the FCW containing S/N is inaccessible; the only way to enter system mode is through execution of a system call (SC) instruction.

The refresh and program status area pointer (PSAP) control registers and the system mode stack register are all inaccessible from normal mode. The normal mode stack register is accessible from system mode under the alias normal stack pointer (NSP), so that normal mode programs can pass arguments to system mode programs on the normal mode stack. When the S/N bit is in the normal state, privileged instructions—ie, I/O, interrupt return, nonmemory synchronization, control register manipulation, and halt—cannot be executed; operating system tasks are executed in the system mode.

Another protective feature is associated with the S/N bit. There are two copies of the implied stack register, one for interrupt and one for subroutine returns. One is used when the CPU is executing 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, which is accomplished through a privileged instruction or automatically in response to an interrupt or trap. Privileged instructions are load from control register (LDCTL), interrupt return (IRET), and load program status (LDPS). A system call trap, which is a 1-word instruction with eight programmable bits, allows a normal mode program to call one of 256 system mode programs.

The arcade game illustrates how system and normal modes can be used. All of the application software processes seen in Fig 2 can run in normal mode, while the operating system elements in Fig 3 can run in system mode. Calls to the operating system elements from the applications software processes are made using the 256 system calls. For example, the defender guns process can execute the instruction SC #createprocess in order to fire a rocket. The constant, createprocess, is a number from 0 to 255 encoding one of the system functions—namely, the one that creates processes. Programs and data that constitute the initial state of the new process can be passed to the process creation program in registers or on a stack.
Fig 4 Z8000 system/normal operation. S/N bit of flag/control word determines execution mode, system or normal, of CPU.

Memory management

Existence of a user mode and privileged instructions does not solve the entire protection problem; the other half of the solution involves restriction of memory use. Most CPU designs call for a comprehensive memory management facility to unify the approach to restriction of memory use, memory mapping, program relocation, sharing of programs and data, and stack use.

The Z8000 uses an external memory management unit (MMU) that is integrated with a segmented addressing scheme in the CPU. The MMU translates addresses, checks attributes, and interrupts the CPU if an invalid access occurs. Sets of attributes are checked against access rights implicitly or explicitly associated with each process. Then, for example, if a program in user mode attempts to access a memory address whose attributes do not match the program’s access rights, the CPU will trap to a system routine designed to deal with such invalid accesses. CPU addressing scheme and the MMU determine which sets of attributes can be associated with portions of the memory address range. Typically, attributes are associated with a segment in a machine that uses 2-dimensional, or segmented, addressing. In a machine with linear addressing, attributes are usually associated with fixed size blocks of addresses called pages.

The arcade game probably does not need memory mapping or virtual memory, since the total memory space of such an application is small. Access restriction, relocation, and sharing of programs and data can be useful in any application, however. On the other hand, UNIX and UNIX-like operating systems, in which there are many small processes, are well suited to the Z8000’s segmented addressing and memory management.

Use of stacks

Stacks are important tools for meeting the operating system’s responsibilities. A stack is a last in, first out memory associated with two operations: pushing (adding an item) and popping (removing an item). Stacks are explicitly or implicitly used by the operating system to allocate memory in a flexible way, which, in connection with based addressing, allows programs needing non-register storage to be reentrant and position independent. A special case of this is storage or return addresses for subroutine calls and machine state for interrupt processing. In the arcade game, the use of stacks to allow reentry of programs plays an important role. Rocket processes, for example, can all share a common processing routine while each uses a different set of data.

Z8000 architecture calls for the placement of stacks as arrays in memory with an address register marking the top of the stack and providing, through based addressing, access to items at locations relative to the top of the stack. The stack register is a dedicated (special purpose) register in some architectures. In the Z8000, any of the registers R1 to R15 can be used as a stack register, although the architecture determines 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 provision for overflow and underflow protection difficult. The Z8000 provides stack limit protection through use of the attribute specification associated with memory protection. Other architectural features are desirable for the support of stacks, including the ability to designate one or more stacks for program use, single- and multiple-argument push and pop instructions, and automatic warning (traps) of impending stack overflow or underflow.

Context switching

One difficulty that arises when several processes run concurrently is the overhead associated with context switching. The context of a process is that portion of its state which occupies shared resources. For example, since all processes must share the program counter (PC), each process’s PC value is part of its context. The Z8000 has a single set of general purpose registers, control registers, CPU status registers, and so forth. Thus, when the same processing element (CPU) is allocated to more than one process, the process contexts must include the contents of any register that is used. Context switching saves the context of one process and recalls the stored context of another process.

Automatic context switching is provided for interrupts and traps. When an interrupt occurs, the current CPU status (FCW and PC) is saved on the system mode stack, along with a “reason” read from the address data lines AD15 to AD0 during the interrupt acknowledge cycle. Then new values for the FCW and PC are taken from the program status area (PSA). The IRET instruction restores PC and FCW to the preinterrupt state and discards the reason, leaving the stack as it was before the interrupt. Architectural features that expedite context switching include automatic saving of CPU state on interrupts, single-instruction block register saving and restoring, and access to all necessary control registers.

The Z8000 interrupt and trap handling facility provides an automatic, rapid context switch from the executing program to the interrupt processing routine using interrupt vectors stored in a memory table (the PSA). The FCW, PC values, and a reason are saved on the
system mode stack, and new FCW and PC values are set from the PSA entry (vector) corresponding to the interrupt type. The IRET instruction restores the CPU to the preinterrupt state, while at the same time removing the saved information from the stack.

Context switching involving general purpose registers is facilitated in the architecture by block register saving and restoring instructions. These can be used to simulate pushing or popping a block of registers to or from any stack. For example, the eight registers R0 to R7 can be saved on the stack controlled by register RR14 by executing

DEC R15,#16 !Make room on stack!
LDM @RR14,R0,#8 !Save the registers!

These two instructions require 39 clock cycles of execution time, or less than 4 μs at 10 MHz.

**Stacks are an important tool for meeting the operating system's responsibilities.**

In some cases, the values of control registers are essential to the context of a process; the normal mode stack register and the flags register, which contains the bits that define condition codes such as "less than or equal to," are obvious examples. A load control register instruction allows the transfer of any of these registers to or from a general purpose register, permitting them to be saved and restored.

**I/O system and interrupts**

Operating system responsibilities in the I/O system and interrupts vary greatly with the type of application. Architecture of a general purpose CPU must provide the flexibility necessary to accommodate the I/O requirements of a wide range of applications.

One of the operating system's most difficult tasks is 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.

Architectural features that support the I/O system and interrupts are a vectored interrupt scheme; specification under program control of the CPU state to be established for each type of interrupt; and a rapid, automatic context switching mechanism in response to interrupts. Also desirable are a means of defining conflict resolution policies and interruptibility of interrupt processing; a coherently designed family of components, compatible interconnection bus, and established set of bus protocols to allow future family growth; block I/O instructions and direct memory access; and 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. This is done on the basis of either the port of connection or the contents of a vector supplied by the interrupting device.

The PSA block of memory stores interrupt vectors (ie, the new CPU status) for each type of interrupt and trap. In addition to separate lines for nonvectored and vectored interrupts, as well as a nonmaskable interrupt for situations that cannot wait, there is a table of PC values to be indexed by an 8-bit vector placed on the AD bus by the interrupting device. The block of memory used for the PSA is not fixed, as it is in 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 achieved through a simple scheme. The three levels of interrupt—nonmaskable, nonvectored, and vectored—are assigned three levels of priority by the CPU. Using the privileged disable/enable interrupt (DI/EL) instruction, the vectored and nonvectored interrupt lines can be masked so that interrupts wait until the unmasking of the associated line. When interrupts arrive simultaneously on more than one line, priority determines which will be processed first. The processing routine for one interrupt type can be interrupted by the routine for another 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 interrupt sensitive code between DI and EI instructions.

A priority scheme is daisy chained through devices attached to the CPU on the same interrupt line. In this way devices closer to the CPU can interrupt the processing of more remote device interrupts unless the given line is masked during all or part of the processing. This approach allows any priority resolution scheme to be implemented externally.

Block I/O instructions and direct memory access are important and straightforward performance improvement features. Block I/O instructions require careful implementation; they must use general purpose registers continuously to save their current state so that they can be interrupted. Direct memory access functions require the development of bus control protocols and a means of protecting partially loaded or saved memory blocks from access by concurrently executing programs. A key aspect of the Z8000 I/O system is the protection privileged instructions provide, allowing an operating system to manage the I/O interfaces without interference from normal mode programs.

**Distributed control**

When processes to which separate processing units may have been allocated share a common memory, guarded commands and semaphores are used. Basic architectural support for these techniques is atomic test and set (TSET), a CPU instruction that tests a memory location for the value "available" and simultaneously sets the value to "not available." "Atomic" refers to the fact that there can be no other access to the given memory location between the test and set portions of the instruction. This prevents two concurrently running processes from finding the location set to "available" simultaneously.

Architecture provides synchronizing procedures, both for processes that share memory and for those that do not. In the case of shared memory, the TSET instruction
provides the basis for synchronization. In the case of nonmemory synchronization, the Z-BUS specification includes a set of lines and a protocol for resolving simultaneous requests for shared resources while the CPU provides instructions to support the bus connection and protocol.

Support for conventions
In the design of a CPU, consideration must be given to whether architecture should support all conventions equally or encourage specific conventions through special features. For instance, should a CPU be designed with general support for high level languages, or should it be designed to optimize Pascal at the expense of FORTRAN programming efficiency? Should it provide special features that make a subroutine argument passing convention using the stack especially efficient at the expense of the efficiency of other argument passing conventions? Z8000 design supports many conventions, including a segmented addressing scheme, message passing for interprocess communication, component and backplane bus protocols, and interrupt protocols for all components.

A message is a set of characters (or words) emitted 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. Message passing support includes block I/O instructions in the Z8000 CPU; asynchronous interprocessor connection in the Z-FIO (first in, first out) buffer chip; acceptance of commands from and delivery of messages to the master CPU in designated message registers by the universal peripheral controller (Z-UPC); and allowance for high speed direct access to memory from external devices (e.g., a Z-FIO chip) through the direct memory access chip.

Summary
Several kinds of architectural support are available to system designers for meeting the requirements of the modern operating system. 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, and distributed control and support for conventions are all tools that can expedite effective system resource management.
The performance of two addressing mechanisms on three different microprocessors is examined. One of the mechanisms—and one of the micros—provided superior performance.

A Performance Comparison of Three Contemporary 16-bit Microprocessors

Martin De Prycker*
University of Ghent

The choice of a new computer system is influenced by considerations of various importance: compatibility with the former system, software availability, cost, maintenance, and system performance. To a great extent, the system's performance depends on the central processor's architecture. To study the performance of a particular architecture, two methods are frequently used. One is that which was used in the CFA project, in which three architectural parameters were defined and compared for a set of machine language routines. The other method consists of measuring the execution times of assembly language benchmarks on different processors, as was done at Carnegie-Mellon and by Nelson and Nagle. Other contributions to architecture evaluation have been made by Shustek, who compared instruction execution times, and by Lunde, who evaluated an ISP description of the processors. However, in order to obtain performance figures with any of these methods, the actual processor, or a simulator, has to be available.

The above-mentioned methods involve comparisons of performance made at a low level; here, I compared the performances of processors executing high-level-language programs. In block-structured high-level languages, a major part of execution time is spent on procedure and block entry/exit. (This has been noted by Batson, Brundage, and Kearns, Tanenbaum, and Blake.) When we also include the execution time of variable addressing, it is clear that a large amount of the execution time of block-structured high-level-language programs is spent on procedure and block entry/exit and variable addressing. The overall system performance is thus strongly influenced by the implementation of the addressing mechanism. Therefore, several variable addressing mechanisms have been proposed, e.g., the display mechanism introduced by Dijkstra and the addressing mechanism presented by Tanenbaum.

In a recent paper, I analyzed a method for describing variable addressing implementation performance, one that employs three independent parameter sets: a set of program statistics determined by high-level-language benchmarks, a set of architectural parameters based on the processor architecture and the variable addressing mechanism, and a set of technology-dependent parameters. The usefulness of this model lies in the independence of the three sets, and in the fact that the processor is available in neither physical nor virtual (i.e., simulated) form. Hence, a complete performance analysis can be done analytically. In addition, in order to evaluate the program statistics, the high-level-language benchmarks can be run on any computer system.

Using this analytical model, I compared the addressing mechanisms implemented on a number of processors. I chose three comparable 16-bit micros—the Intel i8086, the Zilog Z8000, and the Motorola MC68000. In the next section I will explain the performance model, as adapted to processors with an instruction prefetch pipeline. I describe a set of Algol and Pascal benchmarks in the third section of this article and

*Now with Bell Telephone Manufacturing Company, Antwerp, Belgium
Addressing mechanisms that implement the block structure in high-level languages

In block-structured high-level languages, program statements can be recursively grouped into composite statements by means of two block delimiters (begin-end and procedure-return). The recursive program structure so generated can be represented by a program tree (Figure 1). Each composite statement or block can thus be given a number, its static lexical level, which is the depth at which the block definition is located in the program tree.

Hence, the lexical level of a block is always determined by the level of the (static) surrounding block: A begin generates a lexical level which is one level higher than the surrounding block; a corresponding end returns the level of the block to the surrounding level. A procedure call generates a lexical level which is one higher than the level at which the procedure is declared; a return puts the level back to the calling level.

Variables may be accessed only when they are declared within the same block or in static surrounding blocks, that is, when they reside at a lexical parent level. With respect to the program tree, this means that we can access all variables declared in path nodes from the root to the actual active node. This also means that scope rules are fully determined by the static program structure known at compile time. Within a block, each variable gets a sequence number, and a lexical address is formed by the pair (lexical level, sequence number). When a block ends (by an end or return), all variables within that block are no longer visible.

For the implementation of the scope rules of a block-structured language, one needs two stacks: a stack with static information (known at compile time), and a stack with dynamic information (known only at run time). Generally, one combines these stacks with the evaluation/allocation stack on which the defined variables and the temporary results are stored. The three stacks are merged into one stack via a linked-list technique. The stack of static and dynamic environments is implemented through marker words that are linked. Among other information, each marker contains two pointers: a static link, pointing to its parent static environment, and a dynamic link, pointing to the previous dynamic environment. The top-most stack marker serves as the base address of the allocation/evaluation stack of the current environment. For the sake of efficiency, the latter stack is implemented contiguously.

It is clear that, with the above simple structure, accessing variables in parent static environments necessitates tracing down the static pointer chain, possibly to a depth of several levels. In order to lessen or avoid this run-time overhead, two mechanisms have been proposed, namely the display mechanism and Tanenbaum's proposal.

The display mechanism. In order to provide fast access to any lexical level, this scheme uses an extra stack (display). Each display location contains a pointer to the base of a visible environment. When a variable at lexical level $i$ is accessed, display[1] is used as base for level $i$. Thus, only one level of indirection is needed to access a variable at any static level. The main benefit of the display mechanism is that the address of any variable can be determined very easily: address = display[i] + sequence number. Thus, the variable access time is independent of the lexical level.

During the execution of statement Q in our example, the display and data stack appear as shown in Figure 2. Variables are accessible through the display: All variables in the three levels can be reached.

Tanenbaum's mechanism. In order to reduce the overhead associated with display rebuilding—which must be done after every procedure return—Tanenbaum reduced the display to two pointers: a local pointer LP and a global pointer GP. Local and global variables can be reached through these pointers, and intermediate variables must be accessed by tracing the static pointer chain through indirections. The rationale behind this approach is that the addressing of variables at levels between the current level and the global level (i.e., intermediate variables) is a relatively rare event.

In our example the data stack during the execution of statement Q will appear as shown in Figure 3. Local (e,f) and global (a,b) variables can be addressed directly; intermediate variables (c,d) can be reached only by tracing the static pointer chain.

![Figure 1. Lexical level and program tree.](image1)

![Figure 2. Display and stack during statement Q.](image2)

![Figure 3. Pointers and stack during statement Q.](image3)

April 1983
discuss their statistical parameters. In the fourth section Dijkstra's and Tanenbaum's addressing mechanisms, as implemented on the three microprocessors, are compared. It is shown that Tanenbaum's mechanism always performs better than Dijkstra's display mechanism. In the last section, I compare the relative performance of the three microprocessors, as a function of memory speed. I conclude by ranking the processors according to their performance. The correspondence with low-level performance analyses performed elsewhere is striking, not only qualitatively but also quantitatively. I also discuss a cost/performance model.

Variable addressing implementation model

In an earlier work, I expressed overall system performance as a function of three independent factors: the high-level-language programs (benchmarks); the processor architecture, i.e., the instruction set and register organization; and the technology. Here, I will examine this model as it has been adapted to processors with instruction prefetch buffers of different lengths.

The overall system execution cost $K$, induced by procedure and block entry/exit and variable addressing, can be written as a product of three independent arrays: one composed of high-level-language program statistics $S$, one determined by the processor's architecture $M$, and one influenced by the technology $K_T$. That is,

$$K = K_T \cdot M \cdot S^T,$$  \hspace{1cm} (1)

where the superscript $T$ denotes array transposition.

This model was obtained in a very straightforward way: The execution cost of any high-level-language program can be determined as a weighted sum of the execution costs of the individual high-level-language instructions, with the frequency of these instructions in the test program as the weight factor. Thus, we can write

$$K = T \cdot S^T.$$  \hspace{1cm} (2)

The array $S$ contains high-level-language program statistics concerning variable addressing, and thus is independent of either architecture or technology. The statistics which make up the $S$ array comprise the following:

- The number of block entry/exists ($n_b$).
- The number of procedure call/returns ($n_p$).
- The number of variables accessed in the program ($n_l$).
- The number of local variables accessed ($n_l$). Local variables are variables which are accessed at the same level at which they are declared.
- The number of global variables accessed ($n_g$). Global variables are variables which are declared at the outermost level.
- The number of intermediate variables accessed ($n_i$). Intermediate variables are nonglobal variables which are accessed at an higher lexical level than that at which they are declared.

The total lexical-level difference of intermediate variables ($d_{ih}$), that is, the sum of the lexical-level differences between declaration and access.

The total lexical-level difference between declaration and access of procedures ($d_{ip}$)

The operations described here can be viewed as "generic instructions," and each high-level-language program can thus be written as a sequence of these generic instructions.

In Equation 2, $T$ denotes an array of execution costs $T_i$ of the generic instructions $i$, or

$$T = (T_1 \ldots T_r \ldots T_n).$$  \hspace{1cm} (3)

One possible description of the execution cost $K$ is the execution time of the test program. Since my study involves only microprocessors, this execution time can be expressed in terms of the number of clock cycles, because of the indivisibility of the clock cycle time $t_i$ (in nanoseconds).

The number of clock cycles $T_i$ needed to execute each generic instruction $i$ depends on various parameters:

- The number of clock cycles $TC_i$ needed to execute each generic instruction $i$. It is assumed that the memory is fast enough (no wait states) and the instruction pipeline is always full.
- The number of extra clock cycles needed to perform a memory read (TMR$_i$) and a memory write (TMW$_i$) and used by slower memory.
- The number of extra clock cycles in the delay TPC$_i$. This delay is caused by an empty pipeline resulting from the execution of a sequence of instructions when not enough memory is free.
- The number of clock cycles in the delay TPS$_i$. This delay is caused by a memory that is slower than specified in the user's manual; hence, extra wait states are introduced in order to have a full pipeline.

The total number of cycles $T_i$ can thus be written as a sum of clock cycles:

$$T_i = TC_i + TMR_i + TMW_i + TPC_i + TPS_i.$$  \hspace{1cm} (4)

The value of each of these parameters is determined by the processor's architecture and technology. If we express each parameter as a product of a technology-dependent part and an architecture-dependent part, then Equation 1 will be satisfied, since the technological parameters are independent of $i$:

$$TC_i = C_i \cdot K_C$$  \hspace{1cm} (5a)
$$TMR_i = MR_i \cdot K_{MR}$$  \hspace{1cm} (5b)
$$TMW_i = MW_i \cdot K_{MW}$$  \hspace{1cm} (5c)
$$TPC_i = PC_i \cdot K_{PC}$$  \hspace{1cm} (5d)
$$TPS_i = PS_i \cdot K_{PS}$$  \hspace{1cm} (5e)
If we define a technological array $K_T$ and an architectural array $M_i$ as

$$K_T = (K_C, K_{MR}, K_{MW}, K_{PC}, K_{PS})$$

(6)

and

$$M_i = (C_i, MR, MW_i, PC_i, PS_i)^T,$$

(7)

then we can rewrite Equation 4:

$$T_i = K_T \cdot M_i$$

(8a)

or

$$T = K_T \cdot M$$

(8b)

If

$$M = (M_1 \ldots M_i \ldots M_n).$$

(9)

Applying Equation 8b to Equation 2 finally leads to the basic model of Equation 1.

For each of the five parameters of Equation 5, the question of whether to separate them into technology-dependent and architecture-dependent parts must be individually determined.

**Execution time in the optimal case.** When the memory is fast enough (no wait states) and the instruction pipeline is full, the total number of clock cycles needed for each generic instruction $i$ is the sum of the number of clock cycles $C_{ij}$ needed for the machine instructions $j$ which compose the generic instruction $i$. These numbers $C_{ij}$ can be easily found in the microprocessor user's manual.

**Influence of slower memory on data memory operations.** The read/write timing diagrams of the typical user's manual give the minimum number of clock cycles needed by the processor to execute a memory read or write. We call these values $m_r$ and $m_w$. Let us denote the memory access time as $x$ (in nanoseconds). The memory is fast enough if $x/t_c \leq m_r$ for a data read—no wait states have to be introduced. The number of clock cycles to be inserted depends on the memory speed, e.g., when $m_r < x/t_c \leq m_r + 1$, only one cycle has to be introduced. The number of clock cycles to be inserted can thus be written as

$$D_r = \max\{0,\lfloor x/t_c - m_r \rfloor\},$$

(10)

where $\lfloor z \rfloor$ denotes the smallest integer greater than or equal to $z$. A similar expression $D_w$ exists for data write operations.

This delay occurs for each data memory operation. The total number of memory operations required for each generic instruction $i$ is the sum of the number of memory operations required for the individual machine instructions $j$ ($R_{ij}$ read operations, $W_{ij}$ write operations).

### Pipeline influence.** The number of clock cycles required for each machine instruction, as described in the user's manual of a microprocessor with an instruction pipeline, is only the number of clock cycles needed to "really" execute the instruction. It is assumed that the instruction word is already prefetched and available in the pipeline buffer. However, since the memory bus is not always free to fill the pipeline, sometimes the pipeline buffer is empty. This causes a delay so that the buffer can be filled before the instruction is executed. Microprocessor manufacturers give a typical value of 5 to 10 percent for this delay, but note that the value can be much higher, depending on the instruction sequence.

To determine this delay $TPC_i$ exactly, the internal microcode of each processor would have to be available. However, since no information on this microcode was available, I used a best/worst-case analysis to determine an upper and lower bound for $TPC_i$.

In the **best case** I assumed that all free clock cycles in one machine instruction were grouped consecutively. For instance, when an instruction needed eight clock cycles and two memory operations of three cycles each, I supposed that the two free clock cycles were contiguous, as shown in Figure 1. Only one cycle needed to be inserted to do the prefetch.

The number of cycles to be inserted for each machine instruction can be determined by using the values of $R_{ij}$, $W_{ij}$, and $I_j$ (the number of clock cycles for that instruction), and a table. One such relation for the Z8000, which has a pipeline length of one word, is shown in Table 1.

In the **worst case** I assumed that the free bus cycles were not grouped, as shown in Figure 2. In this example, two clock cycles have to be inserted. The number of cycles to be inserted can again be determined using a table, as shown for the Z8000 in Table 2.

![Figure 1. Memory operation in the best-case model.](image)

**Figure 1.** Memory operation in the best-case model.

**Table 1.**

<table>
<thead>
<tr>
<th>$I_j$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_{ij}$ + $W_{ij}$</td>
<td>0</td>
<td>2</td>
<td>1</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>1</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

April 1983
Influence of slower memory on the use of a pipeline. When the memory is slower than specified, problems can arise in filling the pipeline buffer during instruction execution. These problems cause a delay TPS, that is dependent on the memory speed v. Again, information on the microcode would be needed to determine this delay exactly, and again I used a best/worst-case analysis to find bounds for this delay.

In the best case I took into account only the instructions Q which have just enough free clock cycles to do the prefetch without delay when fast memory is used. This is a lower bound, since I eliminated the instructions which operate without delay even when the memory is slower, i.e., instructions which have at least one free clock cycle available. The number of cycles to be inserted for these instructions Q depends on the memory speed and is equal to D, (Equation 10).

In the worst case I assumed that every instruction causes a delay of D, clock cycles, except the instructions which use the memory data bus very little and thus have enough free cycles. However, since in principle infinitely slow memory can be used, no instruction will have enough free cycles. Therefore I reduced the minimum memory speed to a practical value. This minimum is obtained for a maximum access time \( t_m \). Thus an instruction which causes no delay in doing a prefetch must have at least Z free cycles, with

\[
Z = \max\{m_r, t_m / \frac{1}{t_c}\} - m_r. \quad (11)
\]

This value is maximum (an upper bound) for a minimum value of \( t_c \). This minimum value \( t_m \) means a maximum processor clock frequency.

Given these descriptions, it is easy to determine the M array for both addressing mechanisms in both the best and worst cases; Tables 3a and 3b show M for the Z8000. It is obvious that only the fourth rows of the M arrays differ in the best and worst cases.

The \( K_T \), \( M \), and \( S \) values can be applied to Equation 1 to obtain a lower bound \( K_L \) for the total number of clock cycles in the best case, and an upper bound \( K_U \) for the total number of clock cycles in the worst case. The total execution time of a test program's block-structured and variable addressing instructions, running on a processor with clock cycle time \( t_c \), will always lie in the range \([K_L \cdot t_c, K_U \cdot t_c]\). This range can be used to compare addressing mechanisms and processors, as described in the following sections.

### Benchmarks and program statistics

Processors and addressing mechanisms are usually more suited to some languages and applications than to others. In a statistical analysis, one hopes to eliminate this bias by considering different languages and applications. In this study, I was limited to two languages, and I considered only a few applications. However, even with applications belonging to totally different domains, the results were almost language- and application-independent, as is shown in the next two sections. In my system, I used HP Algol,\(^1\) a slightly changed version of Algol 60, and Swedish Pascal,\(^1\) a version of Jensen and Wirth's Pascal.\(^2\)

![Figure 2. Memory operation in the worst-case model.](image)

<table>
<thead>
<tr>
<th>ONE MACHINE INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMORY</td>
</tr>
</tbody>
</table>

Table 2.

Number of clock cycles to be inserted in the Z8000 for the worst-case model.

<table>
<thead>
<tr>
<th>( R_H + W_H )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>1</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>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 3a.

M for the display mechanism, implemented on the Z8000 for the best and worst cases.

\[ M_{BEST} = \begin{bmatrix} 85 & 194 & 24 & 48 \\ 3 & 11 & 2 & 2 \\ 4 & 7 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 3 & 6 & 0 & 0 \end{bmatrix} \quad M_{WORST} = \begin{bmatrix} 85 & 194 & 24 & 48 \\ 3 & 11 & 2 & 2 \\ 4 & 7 & 1 & 1 \\ 12 & 30 & 3 & 8 \\ 13 & 31 & 4 & 6 \end{bmatrix} \]

Table 3b.

M for Tanenbaum's proposal, implemented on the Z8000 for the best and worst cases.

\[ M_{BEST} = \begin{bmatrix} 64 & 139 & 14 & 14 & 22 & 18 \\ 3 & 8 & 1 & 1 & 1 & 1 \\ 3 & 6 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 5 & 0 & 0 & 1 & 0 \end{bmatrix} \quad M_{WORST} = \begin{bmatrix} 64 & 139 & 14 & 14 & 22 & 18 \\ 3 & 8 & 1 & 1 & 1 & 1 \\ 3 & 6 & 1 & 1 & 1 & 0 \\ 12 & 24 & 2 & 2 & 4 & 2 \\ 11 & 23 & 2 & 2 & 4 & 2 \end{bmatrix} \]
The programs tested concern nonhomogeneous applications such as numerical problems, compiler construction, and data manipulation. They were written by graduate and postgraduate students. Let us call the graduate students programmers A and B, and the postgraduate students programmers C and D. DIGFD, DIGFP, and DIGFK are numerical programs used for digital filtering and speech recognition, and BUBBLE is a bubblesort; all were written in Algol. The Pascal programs are TREE, a program that generates the syntax tree of a program, and SPLIT, which generates the LR(0)-items and adds the look-aheads in a syntax-analyzer generator. The numerical programs were written by programmer C, TREE and BUBBLE by D, and SPLIT by A and B. Dynamic program statistics obviously depend on their input data. Therefore each program was run several times with different input data.

In order to measure the program statistics as described in the preceding section, I developed a measurement system that can analyze any block-structured high-level-language program and measure any high-level-language program statistic. In the same work, I identified a set of useful statistics. For a comparative study of variable addressing mechanisms on microprocessors, I needed only a few of these statistics, namely those defined in the section above. These statistics, measured for the programs described above, are shown in Table 4.

A comparison of two variable addressing mechanisms

In order to compare the display mechanism with Tanenbaum’s proposal, I applied the $M$ array of each to Equation 1. By doing so, I obtained a measurement proportional to the execution time of programs which implement Tanenbaum’s mechanism, and one proportional to the execution time of programs which implement the display mechanism. As stated in the second section of this article, I was also able to analyze the influence of memory speed on these measurements, for the three microprocessors under both the best- and worst-case models.

To compare the two addressing mechanisms, I calculated $R$, which is the ratio of the execution time of Tanenbaum’s proposal to that of the display mechanism:

$$R = K_{TA} \cdot \frac{\tau_c}{K_{DI}} \cdot \frac{\tau_c}{\tau_c}.$$  \hspace{1cm} (12)

Figures 3a and 3b show this ratio, under both the best- and worst-case models, for an i8086 with a memory fast enough to eliminate wait states. This ratio lies in the range [0.73, 0.86] for Algol programs and in the range [0.57, 0.59] for Pascal programs and is almost independent of program and input data. Both figures show that Tanenbaum’s mechanism really performs better than the display mechanism. The better behavior of Tanenbaum’s mechanism in the Pascal programs is due to the low use of intermediate variables in Pascal, which is a consequence of the ability to compile Pascal programs separately. Figures and results for the Z8000 and MC-68000 are very similar.

A measurement system for high-level-language program statistics

The measurement system we developed has two important features: It is independent of language and it can be adapted to any program statistic. Such a system needs three types of input:

1. a description of the language to be analyzed;
2. some indications of the statistics that must be measured; and
3. a program in the language to be analyzed.

In contrast, language-dependent measurement systems lack Input 1—i.e., the language description is built-in.

Since both the description of the language and the description of the statistics are intimately connected with the syntactic structure of the language, a formal means of describing this structure can be used to describe both the language and the statistics. In our system we used the BNF notation developed by Backus and Naur.

Our measurement system uses the above-mentioned connections between the program syntax and the statistics. The way in which this is done can best be explained by considering the compilation process. A compiler first creates the syntax tree of the program (i.e., by means of a syntax analyzer). Then, this tree is converted to machine code via semantic routines, which generate specific pieces of code for each BNF rule. In a high-level-language interpreter system, the semantic routines directly execute the semantic functions associated with the syntactic construct.

In our measurement system, things are similar: We first construct the syntax tree of the program, using an automatic-construction parser. Rather than defining a semantic routine for each syntax rule, we append one or more software probes to some or all syntax rules. These software probes perform one of the following functions:

1. measurement of static statistics,
2. insertion of write statements in particular places in the test program, or
3. insertion of block delimiters (begin-end) to keep the test program syntactically correct and semantically unchanged.

When the converted test program is compiled and executed, the inserted write statements generate trace files, which will later be analyzed to collect dynamic high-level statistics.


April 1983
Analyzing the influence of processor and memory speed on R, I again drew similar conclusions: R is almost independent of processor and memory speed. Figures 4a and 4b show R for the three microprocessors (each with memory that is fast enough) and for an “average” program, i.e., a program exhibiting the average of the statistics shown in Table 4. We see that the ratio is indeed very similar for the three microprocessors. The influence of the memory speed x (in nanoseconds) on a 12-MHz MC68000 is very small (Figure 5). Similar figures can be drawn for the 8086 and the Z8000. Notice also that the influence of memory on slower processors’ R is still smaller.

Given these results, I concluded that under both the best- and worst-case models, and for all three microprocessors, both languages, all programs and input data, and any memory speed, Tanenbaum’s mechanism results in considerably better performance than that provided by the classical display mechanism. The gain in performance reaches a value of at least 14 percent for Algol programs and 39 percent for Pascal programs.

Comparison of the three microprocessors

To compare the execution ties of procedure and block entry/exit and variable addressing in high-level-language programs running on the three microprocessor systems, I used the model described in the second section of this article. Applying the M arrays for the three processors to Equation 1, I obtained sets of performance figures, one for each processor and one for each addressing mechanism in the best and worst cases, and one for the individual programs. With such figures, one can compare two processors for the different cases mentioned above by examining the ratio of their respective performance values.

In the course of my analysis, I arrived at an important conclusion: The relationships among the performances of the microprocessors are almost independent of program and input data. This conclusion can be deduced from Figures 6a and 6b, which describe the performance of each processor relative to the 8086 worst case (assuming that the memory is fast enough), for Algol programs implementing the display mechanism on the Z8000, and for Pascal programs implementing Tanenbaum’s proposal on the MC68000. The figures for different programs and input data differ by only a few percent. Notice also that best- and worst-case results lie within a reasonable range. Because of this program and data independence, only the results of “average” Algol or Pascal programs need to be discussed below. Average Algol or Pascal programs are as defined in the preceding section.
Table 4.
Program statistics concerning variable addressing.

<table>
<thead>
<tr>
<th></th>
<th>(n_b)</th>
<th>(n_p)</th>
<th>(n_t)</th>
<th>(n_i)</th>
<th>(n_g)</th>
<th>(d_l)</th>
<th>(d_p)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIGF</td>
<td>951</td>
<td>963</td>
<td>71583</td>
<td>6</td>
<td>19331</td>
<td>24690</td>
<td>27561</td>
</tr>
<tr>
<td>DCFK</td>
<td>78014</td>
<td>951</td>
<td>963</td>
<td>6</td>
<td>19197</td>
<td>24690</td>
<td>6</td>
</tr>
<tr>
<td>DIGP</td>
<td>71583</td>
<td>951</td>
<td>963</td>
<td>6</td>
<td>19197</td>
<td>24690</td>
<td>6</td>
</tr>
<tr>
<td>BUBB</td>
<td>78014</td>
<td>951</td>
<td>963</td>
<td>6</td>
<td>19197</td>
<td>24690</td>
<td>6</td>
</tr>
<tr>
<td>SPlT</td>
<td>71583</td>
<td>951</td>
<td>963</td>
<td>6</td>
<td>19197</td>
<td>24690</td>
<td>6</td>
</tr>
<tr>
<td>TReE</td>
<td>71583</td>
<td>951</td>
<td>963</td>
<td>6</td>
<td>19197</td>
<td>24690</td>
<td>6</td>
</tr>
</tbody>
</table>

- \(n_b\) = Number of block entry/Exit
- \(n_p\) = Number of procedure call/returns
- \(n_t\) = Number of variables accessed
- \(n_i\) = Number of intermediate variables accessed
- \(n_g\) = Number of Global variables accessed
- \(d_l\) = Total lexical-level difference of intermediate variables
- \(d_p\) = Total lexical-level difference between declaration and access of procedures

Figure 7a shows the influence of memory speed on the execution-time ratio \(K_{Z8000}/K_{MC68000}\) for an average Algo program, with the display mechanism, implemented on 4, 8, 10, and 12-MHz processors. The same ratio is shown in Figure 7b for Tanenbaum's proposal. Both addressing mechanisms have a better performance when implemented on the Z8000 than when implemented on the MC68000, provided that the memory is fast enough for the processor's clock frequency. With slow memories and high processor clock frequencies, however, the MC68000 performance degrades more slowly than that of the Z8000. Indeed, an MC68000 with a slow memory actually performs better than a Z8000 with a slow memory. This behavior can be easily explained. The Z8000 needs only three clock cycles for a memory operation \((m_r = m_w = 3)\), whereas the MC68000 needs four or five cycles \((m_r = 4, m_w = 5)\). When fast memories are used, the Z8000 can operate at maximum speed and thus execute a memory operation in only three clock cycles. A better Z8000 performance is thus obtained. When slower memories are used, Z8000 performance begins to degrade as soon as a memory operation requires more than three clock cycles. This is in contrast to the MC68000, the performance of which does not begin to degrade until a memory operation requires more than four clock cycles. Thus, MC68000 performance degrades more slowly than Z8000 performance for memory speeds of at least \(3 \cdot \ell\), e.g., 250 nanoseconds for a 12-MHz processor and 300 nanoseconds for a 10-MHz processor (see again Figures 7a and 7b).

Comparing Figures 7a and 7b, we see that the Z8000 is better suited to the display mechanism than to Tanenbaum's proposal, compared to the MC68000. The main reason for this lies in the method of computation of the base address of the lexical level, which is slower in the MC68000. In the display mechanism, this operation is performed at each variable access and thus requires more operations in the MC68000. Again note that the

April 1983
best- and worst-case ratios do not differ much: The exact performance ratio lies between tight limits. Similar figures can be derived for an average Pascal program.

Similar conclusions can be reached in comparing the Z8000 to the i8086 (Figures 8a and 8b). One major difference is striking: The performance of the i8086 is much poorer than that of the MC68000.

Since the i8086 and the MC68000 both need an equal number of clock cycles for a data read \((m_r = 4)\), and since only the number of memory write cycles is different \((m_w = 4\) for the i8086, \(m_w = 5\) for the MC68000), the influence of memory speed on the execution-time ratio \(K_{MC68000}/K_{i8086}\) is very small, as is shown in Figures 9a and 9b. Note also that both processors are equally suited to both addressing mechanisms.

Using the results shown in Figures 7, 8, and 9, I made a global performance analysis and compared my results with those from other studies. To obtain one performance value for each processor, I averaged the performances of all the programs in both languages with both variable addressing mechanisms. I also used average performance values from the studies by other researchers; these values were obtained by averaging the performances of all programs, normalized to equal processor clock frequencies. Figures 10a and 10b show the mean performance ratio of programs analyzed by Nelson and Nagle,\(^6\) by Grappel and Hemenway\(^5\) and adjusted by Patstone,\(^2\) by Hunter and Ready, Inc.,\(^2\) and by Hansen et al.\(^2\) They also show an upper and lower bound for my results. The upper bound is obtained by dividing

---

**Figure 6.** Relative performance of the Z8000 compared to the i8086 worst case, with the display mechanism implemented for Algol programs (a); relative performance of the MC68000 compared to the i8086 worst case, with Tanenbaum's mechanism implemented for Pascal programs (b).

**Figure 7.** \(K_{Z8000}/K_{MC68000}\) as a function of the memory speed \(x\) for the display mechanism on 4, 8, 10, and 12-MHz processors (a) and for Tanenbaum's proposal on 4, 8, 10, and 12-MHz processors (b).
Figure 8. $K_{ZX8000}/K_{IX8086}$ as a function of the memory speed $x$ for the display mechanism on 4, 8, 10, and 12-MHz processors (a) and for Tanenbaum's proposal on 4, 8, 10, and 12-MHz processors (b).

Figure 9. $K_{ZMC68000}/K_{IX8086}$ as a function of the memory speed $x$ for the display mechanism on 4, 8, 10, and 12-MHz processors (a) and for Tanenbaum's proposal on 4, 8, 10, and 12-MHz processors (b).

Figure 10. Relative performance of the MC68000 to the IX8086 as determined in five studies (a); relative performance of the Z8000 to the IX8086 as determined in four studies (b).

April 1983
the best-case results for one processor by the worst-case results for the other. The lower bound is similarly obtained by dividing the worst-case results for the first processor by the best-case results for the second processor. The real performance ratio will always lie in the range defined by these bounds. Note that there is a great resemblance among the studies, even when my performance figures include only the times to execute procedure and block entry/exit and perform variable addressing in high-level-language programs. This proves that the results from an analytical model provide great accuracy.

The results can also be combined to provide a cost/performance analysis. Figure 11 shows a global comparison of the three processors with a set of possible clock frequencies. (We assume that each processor is or will be available with a 4, 8, 10, or 12-MHz clock.) The results depicted are for an average Pascal program having the display mechanism, but similar results will be obtained for an average Algol program and/or Tanenbaum's proposal. Even when programs producing different statistics are used, the results will be similar. Thus, various microprocessor system configurations will yield a relative performance of, say, 3.5: a 12-MHz Z8000 with 395-nanosecond memory, a 12-MHz MC68000 with 445-nanosecond memory, a 10-MHz Z8000 with 380-nanosecond memory, or a 10-MHz MC68000 with 415-nanosecond memory. These systems are for the worst-case model.

By taking a set of processors \( T_k \) with a memory speed \( x_{W_k} \), we can find the lowest-cost configuration, depending on the cost of the processor \( P_k \), the cost of the memory \( M_k \), and the size of the memory \( S \). The processor cost \( P_k \) is a function of the processor type \( T_k \), which is characterized by the manufacturer \( m_k \) and the clock frequency \( f_k \)—thus, \( P_k = P(m_k, f_k) \). The memory cost \( M_k \) is a function of the memory speed \( x_{W_k} \), i.e., \( M_k = M(x_{W_k}) \). Thus, for each possible configuration \( k \) we obtain a cost figure \( C_k \):

\[
C_k = P(m_k, f_k) + S \cdot M(x_{W_k}).
\]  

The lowest-cost processor/memory configuration will have the smallest \( C_k \).

Since we used the worst-case model to obtain the memory speed \( x_{W_k} \), we can be sure that the relative performance will be at least minimally acceptable, since the real performance value will always lie in the range [worst case, best case]. Systems using memories with a speed \( x_{B_k} \) obtained under the best-case model can also have the same performance figure, even with a slower memory, since \( x_{B_k} \geq x_{W_k} \). For instance, a relative performance of 3.5 can be provided by a 10-MHz MC68000 and a memory with access time of 540 nanoseconds (>415 nanoseconds), if the best-case results are taken. Since the memory is slower, the cost will be lower. However, given a memory speed \( x_{B_k} \), it cannot be guaranteed that the performance will actually have the value in mind, since the figures are obtained under best-case models and the real performance value can thus be smaller. The choice of memory speed depends on whether the application is time-sensitive. If it is, the worst-case speed \( x_{W_k} \) must be used to ensure that the desired performance will be obtained. If the application is cost-sensitive rather than time-sensitive, the best-case speed \( x_{B_k} \) must be used, since it always results in a cheaper configuration than if the worst-case speed is used. Of course, this approach cannot ensure that the desired performance will be obtained.

We have analyzed the performance of addressing mechanism implementations for block-structured high-level languages. The performance measure defined here can be written as a (scalar) product of three arrays, each array depending on one parameter set. These three sets are completely independent—that is, they comprise technological, architectural, and program-statistical sets.
This model provided a basis for comparing, in three contemporary 16-bit microprocessors, the implementation of the traditional display mechanism to the implementation of the mechanism proposed by Tanenbaum. A best/worst-case analysis overcame the lack of information about the microcode and its relationship to instruction prefetch behavior.

The performance figures presented here were consistent with one another and with those derived in other studies. They showed that Tanenbaum's proposal provided a uniformly better performance than the display mechanism. The figures also indicated the relative performance of the three microprocessors—the Z8000 did the best, the MC68000 the second-best, and the i8086 the worst. These results agreed well with earlier data. The methods presented here also showed how to determine the influence of memory speed on performance, and how the results could be used to obtain a cost/performance figure.

Acknowledgment

The author wishes to thank Dr. J. Van Campenhout for his many helpful comments and for his thorough proofreading.

References


April 1983

Martin De Prycker is a systems engineer with Bell Telephone Manufacturing Company, Antwerp, Belgium, where he is involved in long-range development. A member of the ACM and the IEEE, he received the MS in electrical engineering in 1978 from the University of Ghent, Belgium, and the BS and PhD in computer science from the same university in 1979 and 1982.
A paged-memory management chip brings virtual memory to two 16-bit CPUs. Additionally, a coordinated bus structure makes possible distributed-processing or multitasking, multi-user systems.

16-bit μPs get a boost from demand-paged MMU

Faced with applications that demand large programs and extensive data manipulation, microcomputer manufacturers are turning to virtual memory management, an approach originally developed for minicomputers. A single chip uses demand-paged virtual memory to expand the already large memory-addressing capabilities of two new 16-bit microprocessors.

Running the software being developed for those processors—the 8-Mbyte Z8003 and the 64-kbyte Z8004—means using the latest techniques for effective memory management. The technique known as demand-paged virtual memory, chosen for the Z8015 paged-memory management unit (PMMU), keeps the most frequently used codes in fixed-length blocks in RAM, swapping them in and out of disk storage to extend the range of addresses. Such a scheme naturally leads to multitasking and multiuser systems, since the time spent accessing a disk can be used for other tasks. With the Z8015, for example, the Z8003’s 8-Mbyte logical address space translates into a 16-Mbyte physical address space.

The Z8015 has the same address translation and access protection features as the Z8010 but is based on 2-kbyte pages rather than the variable-length segments used in the earlier chip. Together, the Z8015 and the Z8003 (or Z8004) bring multitasking and multiuser capabilities to the microcomputer.

In addition, the Z8015’s access validation feature protects memory from unauthorized or unintentional access. The memory management unit also generates an Instruction Abort signal during page faults and at the same time saves sufficient status and information to restart or resume any instruction after the fault is corrected.

One important application of virtual memory is in disk-based multitasking systems. A system of this type can be implemented easily with the Z8003 and the Z8015.

Virtual memory enables a system to execute programs that do not fit into its primary memory. In order to accomplish this, a secondary storage device—usually a disk—is required. When a disk access is required, however, the program in progress must be interrupted. This interruption can cause large and unpredictable delays known as paging overhead, which may become excessive because of the slow access time and transfer rates of floppy disks. For a typical personal computer or a small business computer, these delays might slow a system sufficiently to make virtual memory management impractical.

Hard-disk systems, on the other hand, are faster; therefore, the paging overhead will be shorter and...
therefore acceptable. When a CPU must access a rigid disk fairly often—a condition called thrashing—even the comparatively fast disk can produce too much delay.

Fortunately, the paging overhead of a virtual memory can be minimized with multitasking operating systems that allow one task to run while another waits for access to the disk. Such multitasking operating systems can be single-user systems, like MP/M, or multi-user systems, like Unix.

Virtual memory and multiprocessors

A distributed processing system—such as a local-area network or an intelligent terminal—places computing power and data where they are used, rather than at a central host computer. Supplying each processor in such a system with its own semiconductor or magnetic memory would be prohibitively expensive. Virtual memory management, however, permits resources to be shared among all the devices in a system.

The entire Z8000 family, which uses extensively programmable VLSI components, is geared to distributed processing strategies. Furthermore, a variety of features built into the Z-Bus—the interconnection protocol that all Z8000 family components are designed to use—reduces the chances of bus conflicts and data collisions while multiple processors are being employed.

One such feature is the Bus Lock Status signal that accompanies a Test and Set instruction in the Z8003 or the Z8004. That instruction prevents access to a shared memory by another CPU or DMA controller. In that way, two CPUs, using a flag (semaphore) stored in shared memory, keep track of which processor currently has access to a resource. The Bus Lock Status lets other potential bus masters know that a resource is about to be requested.

The Test and Set instruction consists of two separate bus cycles: a memory read, followed by a memory write (Fig. 1a). When asserted, the Bus Lock status replaces Data Read during both cycles (Fig. 1b).

Given the general picture of how the Bus Lock Status is used to implement semaphores, the question of what applications can benefit from the distributed processing approach still remains. One answer is peripheral controllers.

Software and memory management

Most complex peripheral devices are governed by microprocessor-based controllers, and it is natural for a controller CPU and the main CPU to communicate through a shared memory. In such a configuration, semaphore locations can be used to manage access to message buffers, with the Bus Lock Status being used to generate these semaphores.

![Diagram](image_url)
In addition to controlling access to shared resources, another aspect of virtual memory management is handling faults: CPU requests to those memory locations which are not in the physical memory space.

Every memory management scheme involves translating logical addresses into physical addresses. Additionally, most schemes involve both access checking—to prevent invalid accesses—and usage recording to assist in implementing memory allocation algorithms.

For example, consider the flow of control in a simple virtual memory system. During the execution of the main program, if the CPU issues an address that does not correspond to a physical memory, the memory management unit attempts a logical-to-physical memory address translation. At this point, the microprocessor's Wait input is asserted and the memory management circuitry performs the necessary actions, including all disk accesses. Afterward, execution of the interrupted instruction resumes.

There are, however, drawbacks to this approach. First, the CPU is idle while the fault is processed and must therefore be isolated from the bus if direct memory access is used for memory management. Second, the entire fault-processing action is carried out by the memory management circuitry, without help from the CPU.

In an alternative approach that is employed by the Z8003 and Z8004, page faults are processed by the CPU's ordinary interrupt-handling mechanism.
Computer System Design: MMU for 16-bit µPs

(Fig. 2), which generates an Instruction Abort signal. The signal terminates the instruction that has produced the fault before the contents of any registers are changed. After the fault is corrected, the instruction can simply be restarted.

Because certain instructions perform multiple memory transfers, a fault may occur that requires more than a simple restart. For this reason, the Z8015 is designed to monitor the execution of instructions and to provide accurate restart information to the fault-processing routine. Thus, the fault-processing software restricts itself to correcting the fault and resuming execution. Here again, a benefit of multitasking is in switching tasks when a page fault is being processed—allowing another task to run while the necessary disk accesses are in the process of being carried out.

Multiprocessor systems

Not all multiprocessor or multitasking systems are as complex as the one just described, nor are they all shared-resource designs. Some coprocessor systems, for example, have been designed to run Z80 software in systems based on microprocessors like a 6502, 8088, 68000, or Z8000.

Taking that approach one step further is a system that uses a Z8003 with a Z80 and Z8015, plus dual-ported memory, to run under both Unix and CP/M (Fig. 3).

Since no memory management is used for the Z80, only 64 kbytes of the memory must be dual-ported. The remainder needs to be accessible only to the CPU. However, with memory management there is no difficulty in extending the design to accommodate a multitasking version of CP/M. In that case, as much memory as is needed in a particular application must be dual-ported.

The system forms the nucleus of a high-end personal computer that runs Unix on the Z8003 and CP/M on the Z80. In operation, a CP/M task is initiated through Unix, and a Unix task accepts an I/O request from the CP/M program running on the microprocessor, carries it out, and signals its completion to the system.

The dual-ported memory is a shared resource and is controlled using semaphore locations in memory. As described above, a Bus Lock Status issued during the read cycle of the Z8003 Test and Set instructions protects semaphore locations from access by the associated Z80 microprocessor.

3. Using multiprocessor features and a shared 64-kbyte dual-ported memory, a Z8003 and a Z80 can form the heart of a CP/M- and Unix-based microcomputer. Such a system would use a Share semaphore and a Message flag in a shared-memory to carry out a handshake.
Computer System Design: MMU for 16-bit μPs

The 64-kbytes of dual-ported memory can run on the Z8003 under Unix. It is controlled by the Share semaphore—a mechanism that can be easily modified to cover multiple blocks of dual-ported memory. The Share semaphore is used only for Z8003 tasks to control access to the CP/M facility (Fig. 4). In addition, a Start semaphore initiates I/O requests, utility calls, and the Done signal that are passed from the Z80 to the Z8003 by means of a message buffer register.

A Message flag is used for handshaking with this buffer. That flag is set by the Z80, which then waits for it to be cleared before proceeding. The Z8003 clears Message before setting the Start semaphore. Thereafter, its principal loop consists of waiting for message to be set, performing the requested task, and clearing Message.

The Start semaphore indicates that the Z80 is executing programs in the shared memory and is set by the Z80 only during its power-on initialization. Following that, the Z80 microprocessor only clears the Start flag. Subsequent setting is done by the Z8003 whenever a Z80 program has been loaded into the dual-ported memory of the system and is ready to run the program's instructions. After executing the program, the Z80 clears the Start flag.

How useful?

<table>
<thead>
<tr>
<th>Circle</th>
<th>Immediate design application</th>
<th>Within the next year</th>
<th>Not applicable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>553</td>
<td>554</td>
<td>555</td>
</tr>
</tbody>
</table>

4. Tasks running on the Z8003 (a) and the Z80 (b) communicate and synchronize their activities through the message buffer, the message flag, and the Start semaphore. The Share semaphore is used only in the Z8003 to allow its tasks to share access to the Z80 and the dual-ported memory.
As memory spaces for microcomputers grow, linear addressing gets cumbersome and error-prone. Segmented addressing solves these problems efficiently, while anticipating 32-bit addresses.

Segmentation advances \( \mu C \) memory addressing

As a memory model, linear addressing has always presented problems for microcomputers. In addition to invalid accesses, traditional micros have faced four major difficulties: accommodating objects whose sizes vary (e.g., stacks or lists); creating and deleting objects dynamically, causing memory fragmentation; relocating objects after the loader has established linkages among them; and sharing objects among otherwise independent processes. All five major problems—which have increased exponentially as systems have grown—can be avoided by using the abstract addressing model provided by segmentation and implemented in the Z8000 CPU and its memory-management unit.

Segmentation organizes the address space into a collection of independent objects corresponding to the largely separate but interrelated objects found in a typical programming situation. This method works for addressing somewhat like a high-level language: The programmer need not worry about the computer memory's physical implementation. Linear addressing, on the other hand, corresponds to a machine language: The model used for the computer's memory is very close to its actual hardware implementation. Examining some memory-addressing tasks that confront programmers will illustrate the trouble with this "machine language" strategy.

In general, a programmer deals with a variety of objects and their interactions. Depending on how "fine-grained" the picture is to be, a programmer could be said to deal with just two objects, the program and the data. Or, at the other end of the scale, he could be said to deal with a multitude of objects—listing separately each instruction and datum. Between these extremes lies the typical programming situation dealing with largely separate objects.

Richard Mateosian, Senior Microprocessor Specialist
Zilog Components Div.
10460 Bubb Rd., Cupertino, CA 95014

Reprinted with permission of Electronic Design, February 19, 1981
Copyright 1981 Hayden Publishing Co., Inc.
but interrelated objects. A chess-playing program, for example, might include:

- Chessboard display program
- Representation of the current position
- Program to generate legal moves
- Routine to evaluate moves
- File of previously evaluated positions
- Handling routines for the previous-position file
- Program to study published games.

This software might run under the control of an operating system, which can also be divided into objects:

- Task scheduler
- Memory allocator
- Secondary-storage interface routines
- Terminal interaction routines
- Process status table
- System stack
- User-process status tables.

Usually, portions of the computer's memory are allocated to each of these objects. A relocating loader might pack the programs together end to end and then allocate fixed areas for data, also end to end, in memory not occupied by the programs (Fig. 1). In the earliest computers, each object received an address directly related to—in fact, 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 wanted to access any of these objects had to use these addresses. As a result, one problem that has always affected linear addressing is invalid accesses.

This hassle occurs even in the smallest systems and on the smallest computer—a program erroneously uses an address as if it belonged to a certain object. 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 will have been damaged (Fig. 2).

Problems stack up

Trouble also crops up with the use of stacks. A common approach in a single-user system is to allocate the lowest memory values to programs and data and the highest ones to a stack, since the push and pop instructions on most computers are designed to make stacks grow “backwards” in memory. The first item placed on the stack is at the highest-numbered address, and the “top” of the stack is at the lowest-numbered address. If program changes cause the program and data areas to expand, less and less remains for the stack. Sooner or later, a stack push will cause the stack to overflow its allotted area and destroy programs or data (Fig. 3). Such problems are often attacked by creating an “envelope” around the accesses in question. For example, instead of using the computer's indexing capability to access arrays directly, the program might call a subroutine that accepts the index and the identity of the array as arguments and returns a validated memory address for fetching or storing. (The routine might handle the actual fetching or storing as well.) In either case, the routine would validate an access by using the array identity as a key to a set of array attributes, including the array's length and location in memory.
In the case of a stack, a similar envelope would be placed around pushes and pops. Rather than use the machine's push and pop instructions, the program would call subroutines for these operations, generating a large software overhead.

Handling Invalid accesses

Another type of invalid access occurs when several programs or sets of data—not necessarily related to one another—share memory locations. As a result, a program’s accesses might be restricted either to its own subroutines and data, or to portions of memory containing data or subroutines that it shares with another program and to which it is only allowed certain kinds of access (such as "read only" or "execute only").

All the discussed software envelopes can be extended to shared-data access, 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 obtain the information needed to make the accesses directly. To guard against such conflicts, hardware solutions such as limit registers have been introduced.

For example, the operating system might set registers defining the limits of a program ready to run at locations 10000 through 19999. In that case, the program is free to make references of any sort, so long as the address used lies within the given range. An attempt to call a subroutine at any higher address, say at location 20000 would result in a "trap," and control would be returned to the operating system.

An envelope around push and pop instructions could detect invalid accesses before they occurred, and provide an alarm—but this is not a solution. Figure 3 shows only one stack that doesn't run out of memory until the entire memory is exhausted. However, if many stacks must be managed, it might be best to assign a small amount of memory to each stack and then expand those that were about to overflow (Fig. 4). If all accesses to stacks go through the envelopes that surround the push and pop instruction, the stack can be "continued" elsewhere in memory. Through this operation, 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 the program using the stack.

Unfortunately, the way in which stacks are ordinarily used is not well suited to this approach. Frequently, a program is allocated a block of stack space, which it then accesses via "based" addressing — i.e., 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" (obtained, for example, from an instruction) to the "base" address in the register. This common practice is incompatible with the existence of gaps in the set of addresses assigned to the stack.

The traditional solution is 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 can be expanded where it is. This approach

4. A PUSH/POP envelope conceals the allocation of the stack into different segments. Lack of such an envelope for based addressing invalidates this scheme.
Segmentation

has two inherent problems. For one thing, moving objects around in memory and keeping the unused memory all in one place increase the processing overhead. For another, all those base addresses for blocks of stack space that the program has in registers or in storage must be exchanged. Save for the most elementary cases, this obstacle is almost insurmountable.

When no memory-management facility is available, the programmer is limited to the static relocation provided by a relocating loader.

Accommodating objects whose sizes vary leads to yet another problem: creating and deleting objects dynamically. It arises even in the simplest single-user systems—for example, "initialization" code might be abandoned after its first execution and the space given to a large data array. Here, too, the difficulties mount rapidly as the system becomes more complex. Because of the difficulty in relocating addresses, objects that should be moved to keep unused memory together often are not. The unused memory soon becomes fragmented, which makes it increasingly difficult to find contiguous blocks big enough to accommodate newly created or expanded objects—even when the total amount of unused memory suffices (Fig. 5).

Up to now, the only "solution" has been to leave management of the assigned memory to the user program. The user is provided with tools like chaining commands and overlay structures in some systems but, by and large, the creation and deletion of objects are simply treated as part of the algorithm implemented by the program.

Relocation is no easy task

After the loader has established links among program parts, it becomes almost impossible to move any of these parts. A hardware solution has been provided at several levels.

Dynamic relocation, which occurs after initial program loading, requires a mechanism that allows actual addresses to be determined at run time. One solution is provided by various kinds of based addressing, usually in the form of relative addressing: Calls, jumps, and loads of program constants are specified by an offset that is added to the actual program-counter value. Data references, too, are made via offsets that are to be added to a stack pointer or other address register. Relocation by based addressing is called "user-controlled" relocation, since the running program controls setting of the stack pointer or of another address register.

From the standpoint of reliability, "system-controlled" relocation is usually a better solution. Its simplest form, memory mapping, is a translation mechanism that converts the addresses used by the running program (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. For example, a translation mechanism for a value set into a base register automatically adds that value to any address used in the program. This approach is similar to based addressing, which, however, uses an explicit reference to the base register in the instruction. In memory mapping, the base register is used to translate addresses completely independently of the program that generates them (Fig. 6).

One natural outgrowth of memory mapping is a mechanism for sharing objects among otherwise independent processes, even though the mapping mechanism must be more sophisticated than a simple base register. If different blocks of logical addresses are mapped independently of one another, a program or data area in physical memory can correspond to different logical addresses for dif-
ferent 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.

**Segmentation offers better solutions**

Memory mapping, which provides the means for dealing with two major problems plaguing linear addressing, ironically must be part of any segmented-addressing scheme, since physical memories are not usually organized in segments. Moreover, all five major problems stemming from a linear-addressing model can be avoided.

The segmented addressing model assigns to each object in the address space a “name” that is really a binary number. Calling it a name emphasizes that there is no relation between objects regardless of any numerical relationship between their “names.”

In the chess-playing example, the chessboard display program could be assigned the name “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 object 1’s linear address space. If this program occupied 2048 bytes, then the addresses within object 1 would range from (1, 0) to (1, 2047). The length of 2048 bytes would be an attribute of object 1 and the mechanism responsible for the interpretation of segmented addresses would cause an appropriate error indication if an address like (1, 2049) or higher were ever used (Fig. 7).

Consider the case of the current-position program—object 2 in Fig. 7. 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 way to refer to items of this array is indexed addressing. The address of the desired item would be specified by giving the array base address of

---

**Note:**

6. Memory mapping becomes simple with a base register: Its "value" is automatically added to the logical addresses.

7. With segmented addressing, the attributes of all objects are known, and error messages prevent an illegal access before it can do any harm.
(2, 0) in one place—say, 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. If the index were 17, then the item address would be (2, 17); the address manipulation cannot affect the object-name portion of the address, only the linear address within the object.

In object 1 of Fig. 7—the display program—the mechanism responsible for address interpretation performs a similar computation for addressing relative to the program counter. If the program contains a branch 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 adding 1264 to 562 would yield (1, 1826).

Preventing invalid accesses

Suppose that a programming error causes the current-position representation array to be addressed with an index value of 257. In a linear addressing scheme, the result would be a reference to the second byte of whatever object follows the current-position representation array in memory. If the legal-move generation program happened to follow the array in memory, half of its first word would be overwritten. With segmented addressing, the mechanism that interprets addresses would discover that (2, 257) is incompatible with the declared length of the array (256 bytes); an appropriate error indication would be generated.

Once the mechanism to check accesses against declared object size has been established, it takes but a small step to add the checking of other object attributes. Problems like protecting one process's data or program from accesses by another process or allowing "read only" or "execute only" accesses to a section of data or program can be solved by checking attributes associated with the objects in question. 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.

This capability is available in the segmented-addressing model built into the Z8001. Its 32-bit addresses contain two fields, the segment-name field and the "offset"; the latter is added to the physical memory address of the segment "base" to obtain the physical address of the element in question (Fig. 8). For example, if segment 5 has a base address in physical memory of 1024, then the physical memory location addressed by the segmented address (5, 26) is 1050, because 1024 + 26 = 1050.

Enter the memory manager

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 computes the actual physical addresses. This MMU can also associate a variety of attributes with each segment, so it can perform the corresponding access checking and generate 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 to the offset. The result is up to 128 segments, each of them presenting a linear address space of 64 kbytes. Furthermore, the external MMU circuit is designed only to translate the uppermost eight bits of the offset; the eight low-order bits are passed directly to the physical memory. Consequently, all segment-base addresses in physical memory must be a multiple of 256 (since the eight low-order bits are zeroes), and the size of a segment—one of the attributes that the MMU checks—must be a multiple of 256 bytes.

One problem with the Z8001's segmentation scheme is that no object can exceed 64 kbytes in size unless it consists of more than one segment. For-
Fortunately, this rather infrequent problem can be solved by software with very little overhead. For example, to access the byte with an index kept in R3 of the array whose base is in RR2, one must replace the instruction

```
LD RL1, RR2 (R4)
```

with the sequence

```
EXB R4  !move high-order index to segment field!
ADD R3, R5  !add low-order index to offset field!
ADCB RH2, RH4  !add (w. carry) high-order index to segment field!
LD RL1, @RR2
```

where RR4 takes the place of R3. These instructions place several segments “end-to-end” and treat the segment name like a number.

However, the MMU implementation has a twofold speed advantage:

1. 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, thus giving the MMU a one-cycle head start on the address translation.

2. The eight low-order bits of the offset, which go directly to the memory untranslated, are the bits needed first by the memory, which enables the memory to get a small head start on the transaction.

As a result, an external MMU circuit entails very little time penalty in memory addresses. 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.

The architectural advantage of the Z8000 family becomes clear by comparing its economical implementation with the method by which a nonsegmented CPU might achieve memory management. Undoubtedly, the approach will take the 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 permits memory mapping and attribute checking, it suffers from many of the problems of linear addressing. In addition, it cannot achieve the overlap of MMU and CPU computational time that is available via the Z8000’s segmentation scheme. The only antidote to the computation overhead of an off-chip MMU for a linear-addressed machine is to design an on-chip MMU; but with the current technology, this approach is likely to require the sacrifice of other features.

One more noteworthy point to be made about the way the Z8001/MMU combination implements segmented addressing concerns the use of stacks. The most difficult problem associated with dynamically expanding stacks involves the correction of pointers into the stack when a stack is moved to another location. Naturally, this problem goes away with memory mapping, since the logical addresses of the locations already used on the stack don’t change when the stack 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.

Consequently, as Fig. 9 shows, a nonfatal stack-warning interrupt occurs when the stack is nearly full—i.e., when an access is made into the last 256 words allocated to the stack. Moreover, the employed method for memory-address computation and size

9. When data begin to fill the top 256 bytes of assigned stack space, a nonfatal warning is generated to prevent possibly destructive overflow.
Segmentation

specification takes into account that stacks grow downward in memory, from the highest addresses toward the lowest.

Segmented vs linear

Just as there are some who argue that higher-level languages are “inefficient” and deny the programmer the total control of assembly-language programming, a few designers adamantly reject segmentation and cling to linear addressing. In fact, their argument has some merit. Just as high-level languages may be inappropriate for very small systems, segmentation may represent overkill in a small memory space. The Z8000’s answer to this problem is to provide segments large enough to accommodate a small application completely in one segment. One of the Z8000’s addressing modes consists only of offsets, so that no references occur outside the 64-kbyte linear address space of one segment. In fact, for such applications, a smaller package is available that lacks the eight pins dedicated to segment-name output and segment-error interrupt input; this smaller version cannot enter the segmented mode of operation at all.

Drawing the line

Where does one draw the line 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? It is a matter of judgment. The Z8000 architecture provides a 16-bit linear address space; in its 23-bit address space, clever, well disciplined programmers can handle unrestricted linear addressing; in its ultimate 32-bit address space, segmentation is undoubtedly 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 Z80; it seemed shortsighted to ask designers moving from 8-bit to 16-bit or 23-bit systems to face one architectural break today and another in a few years (not to mention the huge investment in already-developed software). By developing his system around a Z8000, a designer will not have to face another architectural upheaval when segmentation is introduced—which, if the address space increases to 32 bits, seems inevitable. 

4-52
INITRODUCTIO
This application note explains how a Z8001 CPU, to which at least one Z8010 MMU is attached, is initialized for segmented operation. Described are the specification of the initial CPU status to be established in response to RESET, execution of the first program out of unmapped memory, and initialization of the first, and possibly the only, MMU.

While an attempt has been made to make this application note self-contained, a general familiarity with the Z8001 CPU and the Z8010 MMU is assumed. For further details, the reader is referred to the technical manuals describing these components (Z8000 CPU Technical Manual, document #00-2010-C, and Z8010 MMU Technical Manual, document #00-2015-A).

INITIALIZING SEGMENTED PROGRAMMING
In response to a RESET signal, the Z8001 CPU establishes the CPU status specified in locations 2 through 6 of segment 0 (see Figure 1). Meanwhile, the Z8010 MMU, which is assumed to be connected to the CPU as shown in Figure 2, enters a state in which it passes the SN6-SN0 and AD15-AD0 lines directly through to its A22-A0 address output lines and asserts a 0 on A23. The practical effect of this is that the first initialization instructions to be executed are taken from specific addresses in physical (unmapped) memory.

Operation of the Z8001 CPU in segmented mode depends on the setting of the SEG bit (bit 15) in the Flag/Control Word (FCW) control register. The initial FCW setting is taken from location 2 of segment 0, so the contents of location 2 must have bit 15 set to direct the CPU to enter segmented operating mode.

The example shown in Figure 1 also has bit 14 set. Bit 14 is the S/N bit, which controls the CPU's choice of system or normal mode operation. The setting of S/N bit directs the CPU to enter system mode. The CPU must begin operation in system mode, since the first order of business is to establish an initial setting for the System mode stack register and to initialize the MMU, which requires the execution of privileged I/O instructions.

The initial setting of the EPU bit (bit 13) in the example shown in Figure 1 is 0; if an EPU is present, this bit can be set initially, but it is also possible for the CPU to determine the appropriate setting of the bit as part of its initialization.

The interrupt enable bits (bits 12 and 11) are initially set to 0 by the FCW specified in Figure 1. This is mandatory during the initialization process, because there is no automatic initialization of the System mode stack register; the System mode stack is used in the processing of all traps and interrupts.

The initial PC value of segment 0, offset 8 given in the example in Figure 1 is a convenient one, since it means that the initialization programs can follow the initial CPU status in memory. Also, the CPU status and the initialization program are in the same area of memory, so only a small part of the physical memory address space need be committed to a specific use.

The addresses of the initial CPU status and the initialization program are logical addresses, but at the time of execution of a reset or power-on sequence, there is no assurance that the MMUs have been initialized to perform address translation. The Z8010 MMU, however, has been designed to enter
a mode after a reset or power-on sequence in which it passes addresses directly to physical memory untranslated. (More precisely, it performs a simple, well-defined translation: segment N offset K is translated to physical address \( K + N \times 2^{16} \).) Thus, the initial CPU status is taken from physical addresses 2 through 6, and in the example shown in Figure 1, the initialization program begins at physical address 8. One of the tasks that the initialization program must perform is to initialize MMU mapping tables. Ultimately, the initial CPU status and initialization code can be removed entirely from the logical address space, remaining in physical memory, that can be left inaccessible until another reset or power-on sequence occurs.

Figure 3 shows an initialization program that continues the example begun in Figure 1. The program carries out three steps:

1. Initialize the Stack register (RR14) and Program Status Area Pointer (PSAP) to point at a small temporary stack and a skeleton Program Status Area, both in known locations in physical (unmapped) memory. (The permanent PSA and stack will be established in mapped memory after initialization of memory mapping.)

2. Call the SETMMU routine (Figure 5) to initialize memory mapping, leaving the locations in segment 0 used by the initialization sequence still mapped to the same physical locations they were using before MMU initialization.

3. Initialize the Stack register and PSAP to address the "real" stack and Program Status Area in mapped memory.

After carrying out these steps, the program transfers to the SYSTART routine (not in segment 0) to continue initialization of the specific application. The routine at SYSTART is free to establish a new mapping for segment zero, rendering the initialization code inaccessible; another reset makes it available again.

The routine at STARTUP; the skeleton Program Status Area at INITPSA (Figure 4), and the SETMMU routine and its associated table at MMTAB (Figure

---

**CPU Status for RESET Instruction Memory, Segment 0, Offsets 2-6**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Contents (hexadecimal)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Irrelevant</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>C000</td>
<td>Initial FCW: SEG (bit 15) and S/N (bit 14) set; all others 0</td>
</tr>
<tr>
<td>4</td>
<td>0000</td>
<td>Initial PC: segment 0 (bits 14-8); all other bits must be zero</td>
</tr>
<tr>
<td>6</td>
<td>0008</td>
<td>Initial PC: offset 8 (16 bits)</td>
</tr>
<tr>
<td>8</td>
<td>(Start of startup program)</td>
<td></td>
</tr>
</tbody>
</table>

The values shown are a possible setting for the initial CPU status to be established when a RESET signal is received. The FCW setting is taken from segment 0, offset 2. The value C000 shown here results in the setting of segmented operating mode (bit 15) and System mode (bit 14). Bit 13 is 0, indicating that no EPU is present, and bits 12 and 11 are 0, indicating that neither vectored nor nonvectored interrupts are enabled. The settings of the FLAGS bits (bits 7-2) and the unused bits (bits 1-0) are irrelevant in this example.

The PC segment number and offset are taken from segment 0, offsets 4 and 6, in the standard two-word segmented address format. Any address can be specified. The value of segment 0, offset 8 shown here allows the startup program to begin at the next location of segment 0.

If MMUs are part of the system, they must handle the initial instruction fetches properly, even though the CPU has not yet initialized the MMU translation tables.

---

*Figure 1. Locations 2-6 of Segment 0 Determine Initial CPU Status*
4) all reside in ROM, whereas the temporary stack (which need not exceed 10 words in length as the present program is written) must reside in RAM, preferably in "physical segment 0", i.e., in the first 65,536 bytes of physical memory. In fact, using the MMTAB entry for segment 0 shown in Figure 4, the temporary stack should reside in the first 784 bytes of physical memory. Since all of the instructions and tables shown in Figures 1 through 5 occupy less than 512 bytes, a physical memory whose first 784 addresses refer to 512 bytes of ROM and 256 bytes of RAM (usable later for other purposes) will suffice.

The skeleton PSA shown in Figure 4 needs little explanation. Only the segmentation trap and the nonmaskable interrupt must be provided for, since no other interrupts or traps can occur in the course of executing the programs shown in Figures 1 through 5. (Of course, a memory error could lead to an unimplemented instruction or system call trap, and a faulty CPU could do practically anything.) Both of the interrupt routines provided do nothing but halt. The segmentation trap routine could do something more intelligent if it had access to a means of communicating error information to the "outside world."

The MMU initialization program shown in Figure 5 is easily understood by anyone familiar with the contents of the Z8010 MMU Technical Manual. It begins by transmitting a set of segment descriptors to the MMU; then it enables address translation by the MMU. Two "programming tricks" and a convention must be understood.

This diagram shows the convention adopted in this application note for the connection of the first (possibly only) MMU. This MMU will translate references to segments 0 through 63 (SN6 = 0). Its Chip Select (CS) signal is activated by a 0 on AD1, which means that any special I/O transaction whose I/O address has a lower byte in which bit 1 is zero will be recognized as a command by this MMU. The reason for using the complement of the given A/D line to select the chip is an artifact of the behavior of 3-state logic. The "floating" value shows up as a High on CS during a reset. Allowing the Reset line to be input to CS causes this MMU to pass addresses to the memory untranslated after a reset. In multiple-MMU configurations, the Reset line needs to be tied to CS for only one of the MMUs. MSEN is set and TRNS is cleared in that MMU, allowing it to pass the initial memory accesses untranslated. All other MMUs will 3-state their outputs. The form of connection shown here is the same as for MMU #1 in the examples in the Z8010 MMU Technical Manual (doc #00-2015-A).

Figure 2. MMU Is Connected as MMU #1
The first programming trick is the use of a computation to determine the number of bytes to be transferred to the MMU by the SETMMU instruction. The required number is the difference between the offset portions of two addresses: the first descriptor byte and the first byte past the descriptors.

The second programming trick is the inclusion of the initial SR and mode register values in the table of descriptor values. This programming trick is useful because the two best instructions to perform the one-byte transfers are SOUT and SOUTB. The only alternative to the last two instructions before the RET, for example, is

\[ \text{LDB RHO,} \#\text{X2} \]
\[ \text{SOUTB XVOID, RHO} \]

That alternative is perfectly acceptable in this case, but in cases where the identity of the MMU to be addressed is not known in advance, the alternative shown in Figure 5 is preferable.

The convention that must be understood concerns the way in which the special I/O instructions are used to select MMU operations. The MMU opcode or internal register address is represented in the high-order byte of the special I/O space address, while an MMU selection code (decoded by special circuitry) is contained in the lower byte. In the example in Figure 4, the register R4 contains the special I/O address. The low-order byte (R14) contains the complement of the value 3 (bit 1 clear, all other bits except bit 0 set), which is the selection code for MMU #1. The upper byte (R14) first contains 1 (the "address" of the MMU's internal SAR register), then 2 (the opcode for "transmit descriptor and increment SAR"); then 0 (the "address" of the MMU's internal mode register).

The table at MMTAB (Figure 5) can be easily understood. The first entry, a single byte of 0, is used to initialize the SAR (segment address register), an internal MMU register used to determine which of the 64 segment descriptor registers is being addressed by the command to the MMU.

The next 4*(n+1) bytes are the values used to initialize the descriptors for segments 0 through n. This is done using a block I/O transfer to the MMU "address" that loads a descriptor register (four bytes) and then increments the SAR to address the next descriptor register.

The final byte is used to set the MMU mode register ID field to 0 and the bits MSEN and TRNS to 1; this is a change from the values

\[ \]
established by the RESET: MSEN set, TRNS zero. MSEN (master enable) must be set to enable the MMU to emit addresses (otherwise its address output lines remain 3-stated). If MSEN is set, the TRNS bit determines whether address translation is performed (TRNS = 1) or addresses are passed through as 23-bit patterns (TRNS = 0). The other settable bits of the mode register, which are left clear by the value shown in Figure 4, are URS, MST and NMS. URS (upper range select) allows the MMU to respond to segment numbers 64-127 rather than 0-63 on the CPU output lines SN6-SN3. MST (multiple segment tables) allows selective enabling of address translation by the given MMU (CS is used to enable command recognition by the MMU but has no effect on address translation). If MST is set, then matching the NMS (normal mode select) value with the MMU's N/S input line serves as an enabling criterion for address translation.

Setting the ID field of the MMU's mode register to 0 directs the MMU to respond to the segment trap acknowledge status output of the CPU by asserting AD8 (8 + value of the ID field) and leaving AD15-AD9 3-stated. Using the conventions given in the Z8010 MMU Technical Manual, this identifies the MMU as MMU #1 in the "reason" placed on the stack when a segment trap occurs.

The number and values of the descriptor settings in the table at MMTAB depend on the details of the specific application and are not discussed further here. The additional initialization code at SYSTART also depends on the specific application. Typically, this code initializes peripheral device handling, enables interrupts, and starts user processes. The details are not discussed here.

This concludes the discussion of the specific details common to the initialization of any Z8001 CPU/Z8010 MMU system. Variations are possible, but, in most cases, the general form of initialization shown here is followed.

! This is the Program Status Area used temporarily during the stage of initialization that precedes the initialization of memory mapping. It resides in physical memory directly following the STARTUP routine.

! INITPSA: word 0,0,0,0 !Unused entry!
word 0,0,0,0 !Unimplemented instruction trap!
word 0,0,0,0 !Privileged instruction trap!
word 0,0,0,0 !System Call trap!
word 0,%C000 !Segmentation trap!
address SEGTRAP
word 0,%C000 !Nonmaskable interrupt!
address NMISTOP

! No more of the PSA is required. Processing routines can reside in immediately following locations.

! NMISTOP: HALT
SEGTAP: HALT

This is the bootstrap PSA used for the orderly handling of unexpected interrupts during the phase of the initialization process that precedes initialization of memory mapping. The two processing routines, NMISTOP and SEGTRAP simply halt. More effective actions can be taken in an actual system if appropriate routines exist at known locations in physical memory.

Figure 4. Initial PSA Has Few Real Entries
This is the MMU initialization routine called from the STARTUP program; it assumes a single-MMU system. First, up to 64 of the MMU’s segment descriptor registers are loaded from a table in memory. Then address translation is enabled. The only restriction on the address translation set up this way is that the addresses of STARTUP must continue to be mapped to the same physical locations.

```
SETMMU  LDB RL4,#3  !Select MMU #1 and assure Bit 0 = 1!
       COMB RL4  !Use complement to activate CS!
       LDA RR2,MMTAB  !Address of information for MMU!
       IDA RH4,#1  !Address of SAR in MMU!
       SOUTIB @R4,0RR2,R1  !Initialize SAR!
       LDA RR0,MMTABX  !Next byte past descriptor table!
       SUB R1,R3  !Number of bytes in descriptor table!
       LDB RH4,#0F  !Opcode for descriptor transfer!
       SOTIRB @R4,0RR2,R1  !Transmit descriptor table to MMU!
       LDB RH4,#0  !Opcode for "set mode reg"
       SOUTIB @R4,0RR2,R1  !Enable address translation!
       RET
```

**MMTAB**: byte 0  !Initial value (segment number) of SAR!
word 0  !Segment 0: starts at physical address 0!
byte 2  ! 784 bytes long !
byte %A  ! Execute only !

```
word BASEn  !Segment n (<63): starts at 256*BASEn!
byte SIZEn  ! 256*(SIZEn + 1) bytes long !
byte ATTRIBUTESn  ! attributes as specified !
```

**MMTABX**: byte %CO  !MMU mode register value: MSEN, TRNS; ID = 0!

This MMU initialization routine transmits the table of segment descriptors at MMTAB to the MMU addressed by special I/O instructions with a lower byte in which the value of bit 1 is 0 (MMU #1 using the conventions suggested in the Z8010 MMU Technical Manual). Finally, it transmits a mode register value in which the MSEN and TRNS bits are set and all others are 0.

**Figure 5. A Few Instructions Initialize the MMU**
INTRODUCTION

The Z8001 CPU, which is designed to operate with 8M byte segmented memory address spaces, can also be operated in a nonsegmented mode. Thus the user gets the best of two worlds: the flexibility and power of 8M byte segmented memory address spaces, and the economy of 16-bit addresses. Furthermore, the Z8000 CPU Family has been designed in such a way that operation of the Z8001 CPU in nonsegmented mode is compatible, to the extent possible, with operation of the Z8002 CPU, which is designed to be used exclusively in nonsegmented mode.

This application note first describes in detail the differences in memory and register space requirements and in instruction execution times between segmented and nonsegmented Z8001 CPU operation. It then enumerates and discusses the few points of incompatibility between Z8002 CPU operation and nonsegmented Z8001 CPU operation. The Z8003 CPU is identical to the Z8001 CPU for the purposes of this note.

One of the trickier points in dealing with nonsegmented Z8001 CPU operation is the mixing of nonsegmented and segmented programs within an application. Several ways to handle such mixing are discussed. Finally, to make parts of the discussion completely specific, a means of handling the system call (SC) trap is shown with actual Z8001 CPU programs, and several utility routines designed to be invoked through the SC mechanism are presented.

This application note deals very specifically with "esoteric" details of Z8001 CPU operation. The reader is assumed to have read the Z8000 CPU Technical Manual (00-2010-C) and to be familiar with the general ideas of segmented memory addressing on the Z8001 CPU and with interrupt and trap handling in the Z8001 CPU Family.

ECONOMIES OF NONSEGMENTED Z8001 CPU OPERATION

All Z8001 CPU memory addresses are 23 bits long. In the segmented mode of operation, each address is specified completely, using 32-bit representations in instructions and registers. In nonsegmented mode, all address representations assume implicitly the 7-bit segment number field of the Program Counter (PC), so that only 16 bits are required to represent any address.

The ability to use 16-bit address representations when operating the Z8001 CPU in nonsegmented mode results in economies of both space and time. The economies of space derive from the smaller memory and fewer registers used for 16-bit address representations. The economies of time, generally speaking, derive from the fact that there is no need to fetch or store a second word of address representations in instructions, in registers, or on a stack. Thus, for example, a RET instruction requires an additional three clock cycles of execution time in segmented mode, because an extra word must be popped from the stack. The space and time economies of nonsegmented mode Z8001 operation are summarized in Table 1.
<table>
<thead>
<tr>
<th>Function</th>
<th>Space Economy</th>
<th>Time Economy (clock cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions using direct addressing (compared with full segmented address)</td>
<td>1 word of instruction memory</td>
<td>3 cycles</td>
</tr>
<tr>
<td>Instructions using direct addressing (compared with short segmented address)</td>
<td>----</td>
<td>1 cycle</td>
</tr>
<tr>
<td>Instructions using indexed addressing (compared with full segmented addresses)</td>
<td>1 word of instruction memory</td>
<td>3 cycles</td>
</tr>
<tr>
<td>Storage of an address in a register</td>
<td>1 word register</td>
<td>----</td>
</tr>
<tr>
<td>Moving an address</td>
<td>----</td>
<td>Difference in timing between word and long word version of LD, PUSH, POP, etc.</td>
</tr>
<tr>
<td>CALL or CALR</td>
<td>1 word of stack</td>
<td>5 cycles</td>
</tr>
<tr>
<td>RET</td>
<td>----</td>
<td>3 cycles</td>
</tr>
<tr>
<td>LDPS</td>
<td>2 words of data memory</td>
<td>3-4 cycles</td>
</tr>
<tr>
<td>Loading to or from PSAP or NSP control register</td>
<td>1 word register</td>
<td>7 cycles</td>
</tr>
<tr>
<td>JP using indirect register mode (@) if jump is taken</td>
<td>1 word register</td>
<td>5 cycles</td>
</tr>
<tr>
<td>Use of Indexed addressing to simulate based addressing</td>
<td>Fewer instructions for many operations</td>
<td>2-4 cycles for Load instruction; added savings when shorter programs result.</td>
</tr>
</tbody>
</table>
Table 1 can also be regarded as summarizing the "segmentation penalty" if nonsegmented operation is taken as the standard. It is clear from the table that among common operations the only difference in size between segmented and nonsegmented mode instructions is the extra word required by direct or indexed addressing using full (as opposed to short segmented) addresses in the instructions. Most large programs avoid direct addressing, except for CALL instructions and references to global variables, both of which can use short segmented addressing in a large proportion of cases.

The table also shows that among common operations not involving direct or indexed addressing, the only difference in instruction execution time between the segmented and nonsegmented Z8001 CPU operating modes is in subroutine calling and returning. This difference is due to the saving and restoring of 32-bit return address representations.

A major savings that is difficult to measure quantitatively results from the use of indexed addressing in nonsegmented mode to simulate based addressing. Thus, for example, it is possible to write

\[ \text{ADD R0,}4\text{R15} \]

to add the third word of the stack to the contents of R0. In this construction, the offset (4) plays the role of the address, and the address (the contents of R15) plays the role of the offset. Since each is 16 bits long, there is no difference; they are added together to obtain the 16-bit offset portion of the argument address; the segment number portion is derived from the PC. Thus, based addressing, which is essential for the handling of stack-based data, is available with most instructions.

There is one pitfall to watch for when using indexed addressing to simulate based addressing. Indexed references never result in "stack reference" status on ST3-ST0, since this status only occurs when the Stack register (R15) is used as an address register. In indexed addressing, the address comes from the instruction, and the register contains an offset. Thus, if data and stack memories are distinguished by the ST3-ST0 status outputs, then indexed addressing cannot be used to access stack elements.

**Z8002 Compatibility**

The road between the Z8002 CPU and nonsegmented Z8001 CPU operation is a two-way street: programs can migrate in either direction. For example, a Z8001-based development system can be used to develop and check programs whose target system is Z8002-based. Conversely, a Z8002-based application can be easily evolved into a Z8001-based application by using a nonsegmented Z8001 operation as a first step. Furthermore, utility routines or other parts of a program developed for one of these CPUs could be integrated with programs developed for the other. All of these possibilities illustrate the importance of writing nonsegmented code for the Z8001 CPU.

There are very few differences between Z8002 code and nonsegmented Z8001 code; all of them are associated with interrupt processing (see Table 2).
<table>
<thead>
<tr>
<th>Z8002 Operation</th>
<th>Z8001 Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interrupts and traps, including SC, cause a 3-word CPU status to be saved on the stack in the format:</td>
<td>Interrupts and traps, including SC, cause a 4-word CPU status to be saved on the stack in the format:</td>
</tr>
<tr>
<td>SP ---&gt; reason</td>
<td>SP ---&gt; reason</td>
</tr>
<tr>
<td>FCW</td>
<td>FCW</td>
</tr>
<tr>
<td>16-bit PC</td>
<td>PC - segment number</td>
</tr>
<tr>
<td></td>
<td>PC - offset</td>
</tr>
<tr>
<td>The 256 possible interrupt vector byte values correspond to legal vectored interrupts.</td>
<td>The 128 even-numbered interrupt vector byte values correspond to legal vectored interrupts.</td>
</tr>
<tr>
<td>The Z8002 CPU uses a Program Status Area (PSA) format in which one word is dedicated to each FCW and each PC. No entry is required for the &quot;segmentation trap&quot; vector.</td>
<td>The Z8001 CPU, regardless of the mode in which it is operating, uses a PSA format in which two words are dedicated to each FCW and each PC.</td>
</tr>
<tr>
<td>The Z8002 CPU must be placed in system mode before the IRET instruction is executed.</td>
<td>The Z8001 CPU must be placed into segmented system mode before the IRET instruction is executed.</td>
</tr>
</tbody>
</table>
The practical effect of these differences is very small in many applications. The PSA differs between the Z8002 and Z8001 versions, but the differences are only in the sizes of the vector entries—four words for the Z8001, two words for the Z8002. The Z8001 restriction to even-numbered vectored interrupt devices limits the number of devices to 128, which is ample for most applications. The interrupt and trap routines can be almost identical for the two versions, unless they access the saved PC value or anything "deeper" in the stack. Since the "reason" and the saved FCW are the top two words of the stack in either case, the instructions that access these items can be the same in both versions. The Z8001 versions of the interrupt routines can be written in nonsegmented form. The SEG bit must be set to zero in the corresponding PSA entry's FCW value, and the CPU must be placed into segmented mode before execution of the IRET instruction. A good approach to this is to dedicate one of the SC instructions (e.g., SC #0) to the performance of this kind of segmented IRET. The details of this will be explained in a later section; the advantage of the approach is that it provides a one-word replacement for the IRETs of a Z8002-based program.

When the Z8001 CPU is operating in nonsegmented mode, R14 refers to the same register in both System and Normal modes, just as in Z8002 CPU operation. This is not anomalous or surprising, but many new Z8000 programmers have been confused by the requirement that interrupts be processed in segmented mode. If an interrupt occurs when the Z8001 CPU is operating in nonsegmented System mode, the CPU immediately enters the segmented System mode of operation. At that time, R14 begins to refer to the segment portion of the stack register, and the register previously referred to as R14 is accessible now only by using the LDCTL instruction with the NSPSEG operand. This situation remains in effect until the CPU returns to nonsegmented operation, which could happen before the execution of the first instruction of the Interrupt-processing routine if the FCW loaded from the PSA does not have the SEG bit set.

COMBINING SEGMENTED AND NONSEGMENTED CODE FOR THE Z8001

Segmented and nonsegmented programs can be mixed to any extent desired, since any program running in System mode can carry out the required setting or clearing of the SEG bit in the FCW. If such switching of modes is to be done at many points, or if it is to be done by programs running in Normal mode, two of the 256 SC instructions can be dedicated to the FCW changes.

Programs that access data or call programs in another segment must consist wholly or partially of segmented code. Programs that make no references outside of their own segments can consist entirely of nonsegmented code.

One point to consider when mixing segmented and nonsegmented code is that operation of the RET instruction depends on the mode in which the CPU is operating when the RET is executed, whereas the operating mode on entry to a subroutine is that of the calling program. Thus, special steps must be taken to assure that subroutines called by programs running in either mode behave properly. One approach is to enter such routines through the SC mechanism. Another approach is to allocate two of the SC instructions to subroutine entry and exit functions. The first of these SC instructions is executed as the first instruction of a subroutine to save the caller's operating mode; the second replaces the RET instruction and causes the CPU to enter the proper mode before returning. Furthermore, there can be two versions of the first of these SC instructions; each can save the caller's operating mode, then place the CPU into the mode appropriate for the given subroutine.

A Systems/Application Distinction

One separation of segmented and nonsegmented code is on the basis of the System/Normal operating mode. A set of general utility programs can be written to be executed in segmented System mode, and self-contained application programs can run in nonsegmented Normal mode, using the SC mechanism to make calls on the utility programs. An approach such as this, which centralizes control of the mixing of segmented and nonsegmented programs, avoids the complications of uncontrolled mixing of modes.

THE SC MECHANISM

The preceding discussion includes several references to the use of SC instructions. To allow these examples to be understood at a more concrete level, one of the many possible ways to handle SC traps is elaborated here.

Figure 1 shows a program to be executed each time an SC trap occurs; that is, it is assumed that the address SCHAND will be stored in the PC field of the SC entry (vector) of the PSA. The program at SCHAND is assumed to be segmented, and it accesses the System mode stack, so the SEG and S/N bits must be set in the FCW field of the SC entry of the PSA. Furthermore, the VIE and NVIE bits of the FCW field of the SC entry in the PSA must be 0, for reasons to be discussed shortly.
This SC-handling routine allows each of the 256 SC instructions to be written as if it had its own separate interrupt. An array of 3-word entries called TABLE contains the FCW and PC values to be established for each, except that the VIE and NVIE (interrupt enable) bits in the FCW are taken from the saved status of the program executing the SC instruction.

The Program shown here has not been optimized for speed. Multiplication of the low byte of the reason by 6, for example, can be accomplished in fewer clock cycles than are required for the CLRB and MULT instructions shown here.

**Figure 1. A Flexible SC-handling Scheme**
The program at SCHAND simulates a "vectored interrupt" facility for SC instructions, but the VIE and NVIE values are taken from the saved status of the program executing the SC instruction, not from the "vector" for that instruction. This assures that the routines invoked by SC instructions, which can be called from a variety of priority levels, won't have the side effect of enabling any previously disabled interrupts. For this reason, the FCW entry for SC must leave both VI and NVI disabled.

Given this mechanism, several of the uses of the SC instructions suggested earlier can now be made concrete. Figure 2 shows possible assignments for the first three SC instructions; Figure 3 shows the corresponding TABLE entries and implementing programs. A reader who has difficulty understanding these programs or the program in Figure 1 should review the material on interrupt and trap handling in the Z8000 CPU Technical Manual.

<table>
<thead>
<tr>
<th>SC Instruction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SC #0</td>
<td>Perform segmented IRET</td>
</tr>
<tr>
<td>SC #1</td>
<td>Set SEG bit in FCW</td>
</tr>
<tr>
<td>SC #2</td>
<td>Clear SEG bit in FCW</td>
</tr>
</tbody>
</table>

Figures 2 and 3 show the implementation of the three SC instructions. The program at SEGIRET is operating in segmented mode because of its entry in TABLE, so all it needs to do is return the stack register to its value before execution of the SC #0 and to perform the IRET.

The program at SEGSET implements both the setting and the clearing of SEG. The C bit setting in TABLE distinguishes the two functions. The change to SEG is made in the saved FCW on the stack, which is the source of the status that will be established by the IRET instruction.

Figures 2 and 3 show the implementation of the three SC instructions.
1.0 INTRODUCTION

The Z8000 Calling Conventions allow programs written in various languages for the Z8000 microprocessor to communicate with each other and to share common libraries. The conventions include argument passing, Stack Pointer status, and register assignments on entry to and exit from a routine. The conventions described here apply to all programming languages supported by the Z8000 microprocessor.

Calling conventions were developed that:

- Satisfy the requirements of languages such as C, PLZ/SYS, FORTRAN, and PASCAL.
- Do not introduce undue call and return overhead in code generated by one language processor at the expense of another.
- Minimize the complexity of the code generators.
- Allow passing of structured parameters by value.
- Encourage efficiency by allowing local variables to be kept in registers and parameters to be passed in registers.

The calling convention has three parts which are described in the following sections. These three parts describe:

- How registers may be used by procedures and what happens to the register contents when calling or returning.
- How the stack must be organized when entering, executing in, and returning from a procedure.
- Where parameters must be when entering or returning from a procedure.

2.0 REGISTER USAGE

As shown in Figure 1, the Z8000's general-purpose register set is divided into three groups for the purposes of this calling convention.

![Figure 1. Z8000 Register Usage](image-url)
procedure may use these registers in any way and does not need to restore them to their original values when it returns.

The second group is called the safe registers and consists of RB-R14 for nonsegmented programs and RH-R13 for segmented programs. The values in these registers must be the same when a procedure returns as they were when the procedure was entered. This means a safe register can hold the value of a local variable, because procedure calls will not alter its value. If a procedure changes the value of a safe register, it must save the value of that register when it is entered, and restore it when it returns.

The third group consists of the stack pointer (SP), which is R15 for nonsegmented programs and R14 and R15 for segmented programs. The stack pointer always points to the top of the stack.

The calling convention also allows for, but does not require, the use of a frame pointer to point to the current stack frame (described in the next section). When a frame pointer is used, it is always the highest safe register, R14 for a nonsegmented program, RR12 for a segmented program.

The Z8000 Floating-Point Registers (either simulated in software by the Z8070 emulation package or provided in hardware by the Z8070 arithmetic processing unit) are similarly divided into two groups as shown in Figure 2.

![Figure 2. Z8000 Floating-Point Register Usage](image)

The first group is the floating scratch registers, FR0-FR3. These registers will contain floating-point value parameters upon entering a procedure and floating-point result parameters when returning from a procedure. While executing, the procedure may use these registers in any way and does not need to restore them to their original values.

The second group is the floating safe registers, FR4-FR7. These registers are used in the same way as the general-purpose safe registers and thus the values in these registers must be the same when a procedure returns as they were when the procedure was entered.

### 3.0 Stack Organization

Figure 3 shows how the top of the stack must look when a procedure is entered. The return address must be on the top of the stack (pointed to by the stack pointer), followed by any parameters that must be passed in on the stack. This figure also shows the stack after the same procedure has returned. The only difference is that the return address has been popped off the stack.

![Figure 3. The Stack Upon Entry To and After Return From a Procedure](image)

During the execution of a procedure, the stack will contain a data area called the stack frame (also known as the activation record) for that procedure. The stack frame is allocated on the stack by the procedure and contains saved values,
local variables, and temporary locations for the procedure. Figure 4 shows the stack while a procedure is executing.

![Diagram of the stack during procedure execution]

**Figure 4. The Stack During Procedure Execution**

The called procedure may or may not use the frame pointer as shown. If no frame pointer is used, the size of the stack frame must not change while the procedure is executing. Thus parameters passed in storage by calls from this procedure must be accommodated in temporary locations at the bottom of the stack frame, and not pushed onto the stack. This organization of the stack substantially shortens the subroutine entry and exit sequence.

If a frame pointer is used, then the calling procedures's frame pointer must be saved on the stack by the called routine as shown in Figure 4. If a frame pointer is used, the size of the stack frame can vary, and thus parameters can be pushed onto the stack if desired.

The calling convention allows procedures with and without a frame pointer to be mixed on the stack. From this point of view, the frame pointer is just a safe register that is used in an agreed upon way by certain procedures.

If a procedure modifies the contents of any of the safe registers or floating safe registers while it executes, then it must save the values of these registers in its stack frame when it is entered so that it can restore them when it returns. The highest safe register not used as a frame pointer should be saved at the top of the activation record (nearest the return address) with lower number registers saved at lower addresses. This is the same order used by the LDM instruction. Only those safe registers actually modified by the procedure need to be saved.

Any floating safe registers that are modified by the procedure are saved in the activation record just below the last general purpose safe register. Higher numbered floating registers are saved toward the top of the activation record.

### 4.0 Parameters

Parameters provide a substitution mechanism that permits a procedure's activity to be repeated, varying its arguments. Parameters are referred to as either formal or actual. Formal parameters are the names that appear in the definition of a procedure. Actual parameters are the values that are substituted for the corresponding formal parameters when the procedure is called.

The Z8000 parameter-passing conventions cover three kinds of parameters: value, reference, and result. Value and reference parameters are passed from the calling routine to the called routine. For value parameters, the value of the actual parameter is passed. For reference parameters, the address of the actual parameter is passed. For result parameters, the value of the formal parameter in the called routine is passed to the corresponding actual parameter of the calling routine when the called routine returns.

Each kind of parameter has a length given in bytes (denoted as length(p) for a parameter p). For value and result parameters, this is the length of the declared formal parameter as determined by its type. For languages that do not declare formal parameters or when the procedure declaration is not accessible when the call is being compiled, the length is the same as the length of the actual parameter. For reference parameters, the length is the length of an address, in other words, two bytes in nonsegmented mode and four bytes in segmented mode.
In addition to a parameter's length, the calling convention distinguishes between parameters of floating-point type and parameters of all other types.

The kind, type and length of a parameter are determined by the conventions of the language in which the calling and the called procedures are written. The user must ensure that these conventions match when making interlanguage calls.

4.1 THE PARAMETER REGISTER ASSIGNMENT ALGORITHM

This section describes an algorithm that assigns every parameter in a parameter list to either a general-purpose register, floating-point register, or storage offset. The parameter assigned to a register is passed in that register during a call. A parameter assigned to storage offset is passed in a storage location whose address is the given offset from the Stack Pointer on entry to the called routine. The algorithm assigns as many parameters to general-purpose registers r2-r7 and floating-point registers fr0-fr3 as possible.

The algorithm makes the following assumptions:

There are four kinds of general-purpose registers:

- Byte (denoted as rln, rhn, n = 0...15)
- Word (denoted as rn, n = 0...15)
- Long Word (denoted as rrn, n = 0, 2, 4, 6, 8, 10, 12, 14)
- Quad Word (denoted as rqn, n = 0, 4, 8, 12)

- The length of a general-purpose register r [(denoted length(r))] is 1 for a byte register, 2 for a word register, 4 for a long word register, and 8 for a quad word register.

- Each general-purpose register has a set of underlying byte registers as follows:
  - The underlying register of byte register is the register itself.
  - The underlying registers of a word register (rn) are the byte registers rln and rhn.
  - The underlying registers of a long word register (rrn) are rln, rhn, rln+1, and rhn+1.

- The underlying registers of a quad word register (rqn) are rln rhn, rln+1, rhn+1, rln+2, rhn+2, rln+3, and rhn+3.

This is illustrated in Figure 5:

![Figure 5. The Underlying Registers](image)

- If n > m, general-purpose register rnx or rnx is higher than a general-purpose register rnx or rnx. A byte register rln is higher than a byte register rhn.

- There are eight floating-point registers, fr0-fr7, each capable of holding one floating point value of any precision.

- A floating register frn is higher than a floating register frm if n > m.

The algorithm starts by processing each value or reference parameter in left-to-right order. If there are unused registers of the same size and type as the parameter, the parameter is assigned to the highest of these registers; otherwise, it is assigned to the next available storage location. Once a parameter is assigned to storage, all the parameters in the parameter list that follow it are also assigned to storage. The same thing is then done for the result parameters, except they are assigned to the lowest available registers in sequence r7, r6, r5, ..., r2 (or fr3, fr2, fr1, fr0). The result parameters can overlap value or reference parameters in registers, but not in storage.

The algorithm marks byte registers and floating-point registers as available or unavailable to keep track of which registers have been assigned to parameters, and it uses a variable, current offset, to indicate which storage offsets have been assigned parameters.
4.2 THE ALGORITHM

This algorithm assigns parameters to registers and storage. The phrases in bold are defined in detail in Table A.

1. Mark all byte registers underlying r2-r7 as available, and mark all other byte registers as unavailable. Mark floating-point registers fr0-fr3 as available and mark all other floating-point registers unavailable.

2. Initialize current offset to 4 if in segmented mode or to 2 if in nonsegmented mode (this allows for the return address to which the stack pointer points).

3. For every value or reference parameter in left-to-right order in the parameter list, do the following:
   a. Determine whether p will fit into a register.
   b. If p will fit into a register, assign p to a value/reference register.
   c. If p will not fit into a register, assign p to storage and mark all available byte and floating-point registers as unavailable.

4. Mark all byte registers underlying r2-r7 as available and all other byte registers as unavailable. Mark floating-point registers fr0-fr3 as available and all other floating-point registers as unavailable.

5. For every result parameter in left-to-right order in the parameter list, do the following:
   a. Determine whether p will fit into a register.
   b. If p will fit into a register, assign p to a result register.
   c. If p will not fit into a register, assign p to storage and mark all available byte and floating-point registers as unavailable.

### Table A. Definition of Algorithm Elements

1. **Determine whether p will fit into a register:**
   - If p is a floating-point value or result parameter, then p will fit into a register if there is a floating-point register which is available. Otherwise, p will fit into a register if there is a register r such that length(p) = length(r) and all byte registers underlying r are available.

2. **Assign p to a value/reference register:**
   - If parameter p is a floating-point value parameter then:
     a. Assign p to the highest available floating-point register r.
     b. Mark floating-point register r as unavailable.
   - Otherwise:
     a. Find the highest general-purpose register r such that length(p) = length(r) and all byte registers underlying r are available.
     b. Assign parameter p to register r.
     c. Mark all byte registers underlying r as unavailable, and mark any higher available byte registers as unavailable.

3. **Assign p to a result register:**
   - If parameter p is a floating-point result parameter then:
     a. Assign p to the lowest available floating-point register r.
     b. Mark floating-point register r as unavailable.
   - Otherwise:
     a. Find the lowest general-purpose register r such that length(p) = length(r) and all byte registers underlying r are available.
     b. Assign parameter p to register r.
     c. Mark all byte registers underlying r as unavailable, and mark any lower available byte registers as unavailable.

4. **Assign p to storage:**
   a. If length(p) > 1 and current offset is odd, then add 1 to current offset.
   b. Assign parameter p to storage at offset current offset.
   c. Add length(p) to current offset.
This appendix gives an example of using the Z8000 calling conventions for a C language routine, "caller", which calls another routine, "called".

Figure 6 shows the C code, and Figure 9 shows the corresponding assembly language code. Figure 7 shows the registers upon entry to "called" (just after executing line 25 in Figure 9) and after returning from routine "called" (just after executing line 13 in Figure 9). Figure 8 shows how the stack looks during execution of "called" (line 11 in Figure 9).

```
long called (a,b,c,d,e)
    /*called routine - returns long */
    long b,c;
    int a,d,e;
    { long y;
      return y;
    }

caller () /* calling routine */
    { long a2, a3, x;
      int a1, a4, a5;
      x = called (a1, a2, a3, a4, a5);
    }
```

Figure 6: A Sample C Program

Figure 7. Registers Upon Entry To and Return From Routine Called

Figure 8. The Stack Frame When the Routine Called (From the Sample C Program) is Executing.
1 module MODULE
2 $SEGMENTED
3 CONSTANT
4 fp :=r15;
5EXTERNAL
6 stkseg LABEL !stack segment!
7GLOBAL
8called PROCEDURE
9ENTRY
10 dec fp,#4 !Allocate called's stack frame!
11 ld1 r2,stkseg(fp) !Assign local variable y to return register!
12 inc fp,#4 !Deallocate stack frame!
13 ret
14 END called

15 caller PROCEDURE
16ENTRY
17sub fp,#22 !Allocate caller's stack frame!
18ld r2,stkseg+4+14(fp)
19ld stkseg(fp),r2 !Move a4 to overflow parameter area!
20ld r2,stkseg+4+16(fp)
21ld stkseg+2(fp),r2 !Move a5 to overflow parameter area!
22ld r7,stkseg+4+12(fp) !Move a1 to r7!
23ld r4,stkseg+4(fp) !Move a2 to rr4!
24ld r2,stkseg+4+4(fp) !Move a3 to rr2!
25call called
26ld stkseg+4+8(fp),rr2 !Assign returned value to x!
27add fp,#22 !Deallocate caller's stack frame!
28ret
29END caller

30END modul

Figure 9. Actual Z8001 Code for Program of Figure 4
APPENDIX B

SPECIAL TREATMENT OF FLOATING POINT PARAMETERS

For programs which will run on a Z8000 without a Z8070 arithmetic processing unit or Z8070 software emulator, floating-point value and result parameters should be treated just like non-floating-point parameters.

Until September 1982, all Zilog compilers will pass floating-point parameters in the same way as non-floating-point parameters. Thereafter, the full standard given here will be used.
The Z8000 CPUs are equipped with instructions that allow memory-to-memory transfers to proceed at speeds usually associated with DMA equipment. This application brief shows how to use the two different mechanisms available in Z8000 CPUs for block moves; then it compares their performance for long and short blocks.

The two block-moving facilities in the Z8000 CPUs are the LDIR instruction (and its alter ego, the LDDR instruction) and the LDM instruction. With LDIR, words are moved from one memory area to another at a basic rate of 9 clock cycles per word, using two address registers and a 16-bit counter register. With LDM, words are moved from memory into registers, then from registers into the new memory area. The basic rate for this kind of transfer is 6 clock cycles per word. In either case, there is overhead associated with setup and looping. The differences in overhead make LDM more effective with small blocks and LDIR more effective with large blocks. In either case, only blocks of words, aligned on word boundaries, are considered. For blocks of bytes, there is a byte version of the LDIR instruction but no byte version of LDM.

Figure 1 shows a comparison of the two methods in moving a block of eight words. The method using LDIR requires 88 clock cycles, while the method using LDM requires only 70 clock cycles. At clock rates of 10 MHz, these result in transfer rates of 1.82M bytes per second for the LDIR method and 2.29M bytes per second for the LDM method.

!Assume that RR12 contains the address THERE and RR10 contains the address HERE. The following sections of Z8000 instruction move a block of 8 words from HERE to THERE.

<table>
<thead>
<tr>
<th>Method</th>
<th>Instructions</th>
<th>Cycles</th>
<th>Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDIR</td>
<td>LDK R9,#8</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDIR RR12,RR10,R9</td>
<td>83</td>
<td>88 cycles = 8.8 us @10 MHz or 1.82 M bytes/sec</td>
</tr>
<tr>
<td>LDM</td>
<td>LDM R0,RR10,#8</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LDM RR12,R0,#8</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>70</td>
<td>7.0 us @10 MHz, or 2.29 M bytes/sec</td>
</tr>
</tbody>
</table>

In this case, the LDM version is faster--taking 80% of the execution time of the LDIR version. Other differences are:

1. The LDIR version uses R9 for a counter and modifies RR10 and RR12.
2. The LDM version modifies R0-R7 but leaves all other registers unchanged.

In some applications, the modification of RR10 and RR12 may be desirable, in others it may not.

Figure 1: LDM outperforms LDIR in an 8-word transfer.
Figure 2 shows a comparison of the methods in moving a block of 128 words. In this case the LDIR method is faster, requiring only 1170 cycles as opposed to the 1415 cycles required for the LDM method. At clock rates of 10 MHz, the LDIR method gives a transfer rate of 2.19M bytes per second, while the LDM method achieves a rate of 1.81M bytes per second.

<table>
<thead>
<tr>
<th>Instruction Sequence</th>
<th>LDIR Version</th>
<th>LDM Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R9,#128</td>
<td>7 cycles</td>
<td>7 cycles</td>
</tr>
<tr>
<td>LDIR @RR12,@RR10,R9</td>
<td>1163 cycles</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1170 cycles = 117 us @10 MHz, or 2.19 M bytes/sec</td>
<td></td>
</tr>
<tr>
<td>LD R9,#16</td>
<td>7 cycles</td>
<td></td>
</tr>
<tr>
<td>LP: LDM R0,@RR10,#8</td>
<td>35 cycles</td>
<td></td>
</tr>
<tr>
<td>LDM @RR12,R0,#8</td>
<td>35 cycles</td>
<td></td>
</tr>
<tr>
<td>INC R11,#16</td>
<td>4 cycles</td>
<td></td>
</tr>
<tr>
<td>INC R13,#16</td>
<td>4 cycles</td>
<td></td>
</tr>
<tr>
<td>DEC R9</td>
<td>4 cycles</td>
<td></td>
</tr>
<tr>
<td>JR GT,LP</td>
<td>6 cycles</td>
<td></td>
</tr>
</tbody>
</table>

7 + 16 x 88 = 1415 cycles = 141.5 us @10 MHz, or 1.81 M bytes/sec

In summary, for large or small blocks of data the Z8000 CPUs are capable of effecting memory-to-memory transfers at rates in excess of 2M bytes per second using CPU instructions, without the need for a DMA device.

In this case, the overhead of the loop associated with the LDM version outweighs the speed advantage of the LDM instruction. In fact, even if the LDM version consisted of 16 repetitions of the sequence LDM, LDM, INC, INC (without the INCs on the final sequence), the LDM version would still require 1240 cycles—70 more than the LDIR version.

Figure 2: LDIR outperforms LDM in a 128-word transfer

Assume that RR12 contains the address THERE and RR10 contains the address HERE. Each of the two following sections of Z8001 instructions moves 128 words from HERE to THERE.

<table>
<thead>
<tr>
<th>Instruction Sequence</th>
<th>LDIR Version</th>
<th>LDM Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R9,#128</td>
<td>7 cycles</td>
<td>7 cycles</td>
</tr>
<tr>
<td>LDIR @RR12,@RR10,R9</td>
<td>1163 cycles</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1170 cycles = 117 us @10 MHz, or 2.19 M bytes/sec</td>
<td></td>
</tr>
<tr>
<td>LD R9,#16</td>
<td>7 cycles</td>
<td></td>
</tr>
<tr>
<td>LP: LDM R0,@RR10,#8</td>
<td>35 cycles</td>
<td></td>
</tr>
<tr>
<td>LDM @RR12,R0,#8</td>
<td>35 cycles</td>
<td></td>
</tr>
<tr>
<td>INC R11,#16</td>
<td>4 cycles</td>
<td></td>
</tr>
<tr>
<td>INC R13,#16</td>
<td>4 cycles</td>
<td></td>
</tr>
<tr>
<td>DEC R9</td>
<td>4 cycles</td>
<td></td>
</tr>
<tr>
<td>JR GT,LP</td>
<td>6 cycles</td>
<td></td>
</tr>
</tbody>
</table>

7 + 16 x 88 = 1415 cycles = 141.5 us @10 MHz, or 1.81 M bytes/sec
CHARACTER STRING TRANSLATION: Z8000 vs 68000 vs 8086

Task: Translate a string of 1000 characters from one code to another, e.g., EBCDIC TO ASCII.

EXECUTION TIME (μSEC)
(ALL CPUs AT 10 MHz)

CASE 1: STRING LENGTH IS KNOWN

8086 | 68000 | Z8000
---|---|---
5042 | 3604 | 1404
LINES = 9 | LINES = 7 | LINES = 4
BYTES = 17 | BYTES = 26 | BYTES = 16

CASE 2: STOP IF A SPECIAL CHARACTER IS ENCOUNTERED

8086 | 68000 | Z8000
---|---|---
5606 | 4007 | 2358
LINES = 12 | LINES = 10 | LINES = 9
BYTES = 26 | BYTES = 36 | BYTES = 26
<table>
<thead>
<tr>
<th></th>
<th>Z8000*</th>
<th>68000</th>
<th>8086</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CASE 1:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD R3,#1000</td>
<td></td>
<td>MOVE.L #1000,D3</td>
<td></td>
</tr>
<tr>
<td>LD R6,#STRING</td>
<td>LEA.L STRING,A1</td>
<td>LEA.L TABLE,A2</td>
<td></td>
</tr>
<tr>
<td>LD R8,#TABLE</td>
<td>CLR.L D0</td>
<td>MOVE.B (A1),D0</td>
<td></td>
</tr>
<tr>
<td>TRIRB</td>
<td></td>
<td>MOVE.B 0(A2,D0),(A1) +</td>
<td></td>
</tr>
<tr>
<td></td>
<td>@ R6,@ R8,R3</td>
<td>DBF D3,LOOP</td>
<td></td>
</tr>
<tr>
<td><strong>CASE 2:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LDB RL0,#EOS</td>
<td></td>
<td>MOVE.L #EOS,D4</td>
<td></td>
</tr>
<tr>
<td>LD R1,#1000</td>
<td>MOVE.L #1000,D3</td>
<td>LEA.L STRING,A1</td>
<td></td>
</tr>
<tr>
<td>LD R2,R1</td>
<td>LEA.L TABLE,A2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD R3,#STRING</td>
<td>CLR.L D0</td>
<td>BRA ENTER</td>
<td></td>
</tr>
<tr>
<td>LD R4,R3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LD R5,#TABLE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPIRB RL0,@ R3,R1,EQ</td>
<td>LOOP ENTER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB R2,R1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRIRB @ R4,@ R5,R2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Code and timing applies to Z8001, Z8002, Z8003, and Z8004. For Z8001 and Z8003 in Segmented mode, add five μsec, and four bytes.*
This application note describes the design of a system using a Z8002 CPU and Z-BUS peripherals. This system was designed to demonstrate that a Z8002 system is easy to design and build, and to provide a vehicle for the demonstration and evaluation of Z-BUS peripherals. The system includes:

- Z8002 CPU
- Z-SCC Serial Communications Controller
- Z-CIO Counter-Timer Parallel Input/Output Unit
- Z-FIO FIFO Input/Output Unit
- Z6132 Memory
- 2732 EPROM

Basic goals of this system design were:

- It should be simple, with minimum parts count.
- It should use Z-BUS-compatible components wherever possible.
- It should be expandable

With these goals in mind, the next step in the system design was to select the major devices in the system.

The Z8002 CPU was selected because of its high performance and because its 64K byte addressing range capably handles this application. This allows a system that is hardware compatible with all Z-BUS peripherals and memories, and thus keeps the system cost down.

The peripherals were chosen to demonstrate Z-BUS peripherals currently available (Z-SCC, Z-CIO, and Z-FIO) and because of their ability to support functions necessary for running this system. The Z-SCC provides two channels of serial communications, one for a terminal and one for a link to a host computer, such as the System 8000/Z-LAB. The Z-CIO and Z-FIO are included so that the user of this system will have one of each Z-BUS peripheral available on the board.

The Z6132 memories were chosen because they interface easily to the Z8002 and provide 4K bytes of storage per package. In a simple system such as this, large amounts of dynamic RAM would be overkill. The Z6132 provides all the storage needed in a convenient, easily interfaced device.

The 2732 EPROM was chosen because of its density and speed. The 2732 is twice as dense as a 2716 and is available in higher speeds than the 2716. The higher speed EPROMs would be necessary if this system were to operate at 6 MHz.

The system was designed to allow the use of a modified software monitor from the Z8002 Development Module. Modifying the Software Monitor is accomplished by simply rewriting the serial I/O drivers for connection to a Z-SCC rather than a Z80 SIO, and by rewriting the single-step code, which uses different hardware in the new system. Starting from an existing monitor considerably reduced the time necessary to complete the software.

HARDWARE DESIGN

The Z8000 CPU architecture is based on the machine cycle as its fundamental unit of execution. All hardware interface logic must be aware of what kind of machine cycle is being executed so that, for example, operations intended for memory affect
memory only, and not input/output devices. In order to differentiate between the different machine cycles, logic was included in this system to decode the four CPU status lines, \( S_{T0} - S_{T3} \), and to produce status signals to be used in other parts of the system.

**STATUS DECODING**

\( U37 \) (see the schematics attached to end of application note) is an octal decoder (74LS138) that decodes the first eight status codes (those codes for which \( S_{T3} = 0 \)). Two sections of \( U15 \) (a 74LS00) are used to derive a signal called MREF which is valid for any memory access, regardless of the type of address space (code, data, or stack). MREF is represented by this logic equation:

\[
MREF = S_{T3}(S_{T1} S_{T2})
\]

It would have been possible to include another 74LS138 to decode the upper eight status codes and to OR the three status codes for code, data, and stack memory accesses, but that would have added additional chips, and would have been contrary to the goal of minimum chip count. In addition to this status decoding, one section of \( U15 \) and three sections of \( U16 \) (a 74LS32) are used to generate a signal that is the combination of Data Strobe from the Z8002 and a status signal for stack references. This signal is used to drive the single-step logic, which is discussed later.

**MEMORY INTERFACE LOGIC**

The memory interface logic is divided into two major parts, the RAM interface (for the Z6132s), and the EPROM interface (for the 2732s).

**RAM INTERFACE**

The RAM interface logic consists of even/odd bank decoding, and chip select decoding. The even/odd bank selection is done by one half of a 74LS157 multiplexer (\( U12 \)). It takes as its inputs the byte/word signal (\( B/W \)), the read/write signal (\( R/W \)), and Address/Data bit 0 (\( AD_0 \)) from the Z8002 CPU. For any read operation, both outputs are active. For write operations, if the byte/word line indicates a word write, both outputs are active. For write operations in which the byte/word line indicates a byte write, only the even or odd output is active, depending on the state of \( AD_0 \). In essence, for byte write operations, \( ENA_{EVEN} \) is active if \( AD_0 = 0 \) and \( ENA_{ODD} \) is active if \( AD_0 = 1 \). For any other operation, both outputs are active. This decoding is necessary because, for byte write operations, however, the data appears on both halves of the Address/Data bus, so there must be some way of allowing writes to only one bank of the memory.

The RAM chip select logic is composed of two 74LS138 decoders: one for the even byte (\( U4 \)) and one for the odd byte (\( U3 \)). The decoders have as inputs the uppermost three address bits (\( AD_{15} - AD_{13} \)), the MREF signal decoded from the status lines, and either \( ENA_{EVEN} \) or \( ENA_{ODD} \). Each Z6132 is connected to one of these chip select lines, depending on the address desired and whether it is the even or odd bank device for the address.

**EPROM INTERFACE**

The EPROM interface logic is simpler, because the EPROMs have no requirement for even/odd bank select because they do not respond to write operations. The EPROM chip selection is done by \( U5 \), a 74LS138 decoder. This decoder is enabled by the MREF signal and uses as select inputs \( AD_{15} - AD_{13} \) (the 2732s are 4K x 8 devices). This gives EPROM select signals that allow EPROMs to be placed anywhere within the 64K byte address space of the Z8002. Because there is no even/odd selection, both even and odd byte devices at a given address are wired to the same EPROM select signal.

**WAIT STATE GENERATION**

To accommodate slower memory devices, which are often used for reasons of cost, separate wait state generators are included for the RAMs and for the EPROMs. Each generator takes the chip select signals used on the board and ORs them together. This ORed chip select is then gated with Address Strobe (active High). The resulting signal presets a 74LS74 flip-flop, causing the \( Q \) output to go Low. This signal is used as the wait input to the CPU. The first falling edge of PCLK clocks the flip-flop with the "0" input Low, causing the \( Q \) output to go High again. This allows the generated wait signal to be recognized once, adding one wait state to that memory access. The outputs of both wait state generators go through DIP switches to two sections of a 74LS32, which
combines these wait signals with the BUSY outputs of the Z612s into one WAIT output that is fed to the WAIT input of the Z8002. The BUSY outputs of the Z612s must be included because they may need to generate one or more wait states in order to perform their internal refreshing. The DIP switches allow the user to select one wait state for RAM accesses, EPROM accesses, or both. More elegant wait-state generators are possible with selectable numbers of wait states, but the single wait state circuits were used because of their low parts count and simplicity.

PERIPHERAL INTERFACE

Using Z-BUS-compatible peripherals eliminates all external interface logic except the chip select circuitry. This function is handled by U21 and U6. U21 is used to detect the case in which the upper-most five address bits are all 1s. This signal is fed into one of the enable inputs of U6, a 74LS138 decoder. This decoder is also enabled by the status line indicating an I/O machine cycle. This one decoder gives eight chip select signals derived from the upper eight bits of the Address bus. Because Z-BUS peripherals are byte-wide devices on the low byte of the Address/Data bus, it is wise to perform the chip selection with the bits not used by the peripheral for addressing internal registers. By selecting only on the basis of the upper eight bits, the design avoids conflict with any peripheral, because one device may use the lower six bits while another may use the lower seven bits. To make these chip select signals compatible with other devices, the latched address lines LA0-LA15 are used to drive the decode logic. In this way the chip select outputs are valid throughout the machine cycle. Z-BUS peripherals latch the chip select input on the rising edge of Address Strobe, so a longer chip select signal is not necessary. However, because compatibility with devices other than Z-BUS parts is desirable, and, because using the longer cycle does not add any additional logic (the latched addresses are already needed for addressing the EPROMS), the longer chip select signal was incorporated.

INTERRUPTS

Proper interconnection of Z-BUS peripheral interrupt signals is easily accomplished with the logic already in the system.

The Z-BUS interrupt structure is based on a priority daisy chain for resolving conflicts when several devices interrupt at the same time. In order to allow experimentation with different interrupt input to the CPU (in this case VI, the vectored interrupt input, was used), and the interrupt acknowledge back to the peripherals (VIACK). The interrupt input is a wired-ORed signal, since all peripherals have open-drain outputs for this signal. The interrupt acknowledge output of the status decoder is used to feed all of the peripherals; the priority daisy chain resolves for which peripheral the acknowledge is intended.

SINGLE-STEP LOGIC

The single-step logic is composed of three flip-flops (U22 and U28). The single-step logic is enabled ("armed") by writing to an I/O port address (in this case F900). Writing to this port address sets the first flip-flop (which is connected as a set/reset latch). This then enables the chain of two flip-flops (U28) to count stack operations. Several gates are used to generate a signal valid for any stack reference; this signal is ANDed with Data Strobe.

The instruction sequence for single-stepping is to arm the chain with an I/O write to the single-step port and to follow this instruction immediately with an Interrupt Return Instruction (IRET). The stack has already been set up to return to the next instruction in the user program. The two stack operations in the IRET instruction are counted and a nonvectored interrupt is generated. This interrupt is not generated until the rising edge of Data Strobe during the last machine cycle of the IRET instruction, so it is not recognized during that instruction. It is recognized during the next instruction, which is the next instruction of the user program. This instruction executes to completion, and then the interrupt acknowledge sequence starts.

After one instruction of the user program is executed, control is returned to the monitor. This allows user instructions to be executed one at a time under software control. This method of single instruction execution was used instead of a method that uses hardware control of the CPU so that the monitor could be used to examine and alter memory and register contents between execution of user instructions.

BUFFERING

In the hardware design of this system, an important question was whether or not to buffer
the Address/Data bus and the control signals. Several items were considered in order to answer this question.

When considering the dc loads on the CPU outputs, the only devices that present significant dc loads are the "LS" series devices. A Z8002 output drives at least four LS-series inputs. The memories and peripherals are all MOS devices, and as such have negligible dc loading.

The capacitance of inputs is another item that must be considered. The outputs of the Z8002 are specified at a capacitance of 100 pF, so that the sum of the input capacitances of the devices on the bus must be less than 100 pF. The memory devices have a 5-10 pF input capacitance and the peripherals are typically 10-15 pF. With the number of peripheral and memory devices in this system, there is no problem driving these inputs directly from the Z8002.

Considering the present loading, the status and control signals were buffered by a 74LS244, although Address Strobe, Data Strobe, and read/write also go directly to the peripherals. The status outputs are fed to a number of LS-series devices, so buffering helps the loading here. Status is not critical to timing, so the small delay the buffer introduces has no effect. The Address/Data bus was not buffered so that slower access time memories could be used, but if the system were expanded, it would be advisable to buffer the Address/Data lines with 74LS245 bidirectional buffers.

SOFTWARE DESIGN

The monitor on the Z8002 Small Single Board Computer (SSBC) is a modified version of the monitor used on the Zilog Z8002 Development Module. The commands are the same, except that the TAPE and PUNCH commands have been deleted.

The syntax interpretation for Z8002 SSBC monitor commands is:

<address> := <number_in_16_bit_range>

The following notation is used in the command descriptions:

[ ] Square brackets are used to denote optional quantities, and are not actually to be entered.

| Bar is used to denote "OR." For example, W|B means either of the characters W or B may be used.

(CR) Carriage return.

All commands can be abbreviated to their first letter. Commands and options can be entered in either upper or lower case. All numbers are represented in hexadecimal notation and must begin with a numeric digit. The first character typed on a new line identifies the command being invoked. If the command is not understood, a "?" is printed on the terminal and a new command is requested.

SUMMARY OF COMMANDS:

BREAK <address> [n]
Set and clear breakpoint.

COMPARE <address1> <address2> [n]
Compare memory blocks.

DISPLAY <address> [(# of long words/words/bytes>]
L|W|B
Display and alter memory.

FILL <address1> <address2> <word_data>
Fill memory.

GO
Branch to last PC.

IOPORT <port_address> [W|B]
I/O port read/write.

JUMP <address>
Branch to address.

LOAD <filename>
Load file from host system.

MOVE <address1> <address2> [n]
Move memory block.

NEXT [n]
Step instruction.

QUIT
Enter transparent (terminal) mode.
REGISTER [<register_name>]
Display and alter registers.

SEND <filename> <start_address> <ending_address> [entry_address]
Send file to host system

NOTE
All outputs in monitor mode can be suspended with the XOFF character (CONTROL S), and resumed with the XON character (CONTROL Q).

COMMAND DESCRIPTIONS:

BREAK
Syntax:
BREAK <address> [<n>]

Description:
The BREAK command is used to set a breakpoint at the given even address.

If n is specified, the user program execution is not interrupted until the nth time the breakpoint instruction is encountered. The value for n should be in the range 0001 - FFFF. If n is not given, 1 is assumed. If the BREAK command is issued with no parameters, it clears any previously set breakpoint. This action should be performed before setting the current breakpoints.

When user program execution is suspended by the BREAK command, the monitor prints a message informing the user of the break and the address at which it occurred.

COMPARE
Syntax:
COMPARE <address1> <address2> <n>

Description:
The COMPARE command is used to compare the contents of two blocks of memory.

Locations <address1> and <address2> specify the starting addresses of the two blocks of memory; <n> specifies the number of bytes to be compared. If any locations of the two blocks differ, the addresses and contents of those locations are displayed on the terminal.

DISPLAY
Syntax:
DISPLAY <address> [n of long words/words/bytes] [LWB]

Description:
Displays the contents of specified memory locations on the terminal, starting at the given address, for the given number of bytes.

If the number (n) of long words/words/bytes parameter is specified, the contents of the desired locations are displayed, both in hexadecimal notation and as ASCII characters.

If the number of long words/words/bytes is not specified, the memory locations are displayed one at a time, with an opportunity to change the contents of each location. For each location, the address is displayed, followed by the contents, followed by a space. If the contents at that location must be changed, the new contents are entered at this time. A carriage return, either alone or after the new contents, causes the next sequential location to be displayed.

If the [LWB] parameter is not specified, data is displayed in word format.

A "Q" followed by a carriage return terminates the command.

FILL
Syntax:
FILL <address1> <address2> <word_data>

Description:
The FILL command is used to store the given data word into sequential memory locations starting at <address1> up to and including <address2>. The command addresses must be even hexadecimal numbers.
**GO**

**Syntax:**

```
GO
```

**Description:**

This command is used to branch to the current PC, thus continuing program execution from where it was last interrupted.

All registers and the FCW are restored before branching. Before executing a GO command, ensure that the FCW is set to the appropriate value.

**IOPORT**

**Syntax:**

```
IOPORT <port_address> [W][B]
```

**Description:**

This command is used to read data from the given port address, display the data on the terminal, and write new data to that port address.

After the current port data is displayed, the user can either enter a "Q" followed by a carriage return to terminate the command, or enter a series of bytes or words (maximum 128 characters per line). Bytes or words should be blank delimited with a carriage return at the end. This allows multiple writes to a port without scrolling the terminal screen excessively. If the [W][B] parameter is not specified, byte data is read and written to the I/O port. If a carriage return alone is entered, a zero value is written to the port.

**JUMP**

**Syntax:**

```
JUMP <address>
```

**Description:**

The JUMP command is used to branch unconditionally to the given even address.

All registers and the FCW are restored before branching. Before executing a JUMP, ensure that the FCW is set to an appropriate value.

**LOAD DATA FROM HOST**

**Syntax:**

```
LOAD <filename>
```

**Description:**

This command is used to download a Z8000 program from a host system into the SSBC memory.

The monitor program transmits the command line to the host system exactly as entered. The monitor assumes the host system recognizes this command line. When the SSBC is connected to either a PDS-8000 or a System-8000, this command causes the file <filename> to be opened, the data is converted to Tektronix hex format and transmitted to the SSBC.

The monitor program verifies the two checksum values in each record and stores the data in RAM memory at the address specified in the record. An acknowledgement from the SSBC causes the host to send the next record.

A non-acknowledge from the SSBC causes the host to retransmit the current record up to 10 times, after which a record with an error message is sent and the command aborted.

After successful completion of the loading process, the entry point received in the last record is printed on the terminal. An ESCAPE key is used to abort the LOAD command. Any set breakpoints from a previous program must be cleared before loading a new program.

**MOVE**

**Syntax:**

```
MOVE <address1> <address2> <n>
```

**Description:**

This command is used to move the contents of a block of memory from the source address specified by <address1> to the destination address specified by <address2>. The value <n> is the number of bytes to be moved.
NEXT

Syntax:
NEXT [<n>]

Description:

The NEXT command causes the execution of the next \( n \) user instructions, starting at the current PC, and displays the contents of all registers after each instruction is executed.

The value \(<n>\) should be in the range \%001 - \%FFFF. If \(<n>\) is not specified, 1 is assumed.

QUIT

Syntax:
QUIT

Description:

The QUIT command is used to enter the Transparant mode (terminal mode) from Monitor mode.

In Transparant mode, all keyboard input is passed to the host serial port, and all input from the host serial port is passed to the terminal. The baud rate of the host serial port is controlled by three switches of the eight position DIP switch (U11).

The NMI switch on the SSBC is used to return to Monitor mode.

REGISTER

Syntax:
REGISTER [<register_name>]

Description:

The REGISTER command is used to examine and alter registers.

The following are valid register names:

- Any of the sixteen 16-bit registers named \( R_0, R_1, R_2 \ldots R_{15} \)
- Any of the sixteen 8-bit registers named \( RH_0, RL_0, RH_1, RL_1 \ldots RH_7, RL_7 \)
- Any of the eight 32-bit registers named \( RR_0, RR_2, RR_4 \ldots RR_{14} \)
- Program counter register named \( RPC \)
- Flag and control word register named \( RFC \)

If no register name is given, the contents of all registers are displayed. If a register name is given, the specified register name is displayed, followed by its contents, followed by a space.

If the contents of that register are to be changed, the new contents can be entered at this time. A carriage return, either alone or after the new data, causes the next register.

A "Q" followed by a carriage return terminates the command.

SEND DATA TO HOST

Syntax:
SEND <filename> <start_address> <ending_address> [ <entry_address> ]

Description:

The SEND command is used to transfer the contents of memory of the SSBC to a file on the host system.

The monitor sends the command line to the host system exactly as received. The SEND command on PDS-8000 or a System-8000 opens a file name \( <\text{filename}> \) and sends an acknowledge (ASCII 0) to the SSBC to start transmission.

If the file cannot be opened, an abort-acknowledge (ASCII 9) is sent to the monitor and the SEND command is aborted.

The monitor formats the contents of memory specified by \( <\text{start_address}> \) and \( <\text{ending_address}> \) into Tektronix hex format and transmits this data to the host system. The monitor then waits for an acknowledge before sending the next record.

A nonacknowledge (ASCII 7) received by the monitor causes the same record to be resent up to ten times. If this record is still not sent successfully, a record with double slash characters (//), followed by a carriage return, is sent to the host system to abort the SEND program in the host. The two slash characters are also sent if the ESCAPE key is pressed by the user to abort the SEND process.
The address specified by <entry_address> is sent in the last record as the entry address for that file. If no entry address is specified, an address of %0000 is assumed.

RECORD FORMAT FOR LOAD/SEND COMMANDS:

The record format for the LOAD and SEND commands is Tektronix hex format, which uses ASCII characters only. Each record contains two checksum bytes, a starting address, and a maximum of 30 bytes of data. The format of the record is shown below:

For Records 1 to n:

/<address(4)><count(2)><checksum1(2)><data(2)...<data(2)><checksum2(2)><(CRC)>

For the last record:

This record has a 00 in the count field and indicates the end of the load data.

/</entry_address(4)>00<checksum(4)><(CR)>

<entry_address> The starting address for the program (4 ASCII characters).

<checksum> The checksum for the entry address (4 ASCII characters).

For records with error messages:

If either the host system or the SSBC aborts a LOAD or SEND process, it may send a record of the form:

</error_message_in_ASCII_text><(CR)>

ACKNOWLEDGE:

After each record is received from the host system while loading, an acknowledge (ASCII 0) is sent if the checksum values are verified.

A non-acknowledge (ASCII 7) causes the host system to load the same data record up to 10 times. After the tenth try, the monitor program returns to Monitor mode for the next command, and the host system aborts the LOAD command.

An abort-acknowledge (ASCII 9) is sent to the host system if the user decides to abort the LOAD or SEND process by pressing the ESCAPE key. This action also causes the host system to abort its program. The monitor returns to Monitor mode for the next command.

The address used in the data record during the loading process is specified when the object file is originally created on the host system. This address must be greater than %4500 (%4000 - %4AFF is used by the monitor program).

For the SEND command, data is formatted and sent to the host system in Tektronix hex format. An ASCII 0 response from the host causes the next data record to be sent.

The same data record is sent again if ASCII 7 is received. The SEND command resends the same record up to ten times before it aborts the sending process.

An ASCII 9 response from the host system indicates that the input file already exists, or that an error occurred during a disk access.

MONITOR I/O PROCEDURES:

The SSBC monitor contains subroutines to do character I/O to and from the terminal. These subroutines can be called by a user program in order to do terminal I/O. A description of each
subroutine follows, along with details of which registers, if any, are affected by calling the routines. The hex address in parenthesis next to the subroutine name is address to which the user should do a CALL instruction to use that routine. For example, to output a carriage return and line feed to the terminal, a user should execute the following instruction:

CALL %0FD4 !output CR/LF. RO is lost !

TYIN (%0FA0)
Get a character from the keyboard buffer. If the buffer is empty, this procedure waits for a character to appear. The character is stored in RLO, and the contents of RHO are destroyed.

TYWR (%0FC8)
Display a character in RLO on the terminal. The character is not displayed if the XOFF character is received before this procedure is executed. This procedure waits until an XON character is received to display the character in RLO. If the display character is a carriage return, the zero flag is set and RHO is destroyed.

PUTMSG (%0FC0)
Send a character string to the terminal. Register R2 should contain the address of the character string buffer, and the first byte in the buffer should be the number of characters to be displayed. If there is no carriage return in the string, the entire string specified is displayed, otherwise the string is displayed up to and including the first carriage return. Registers RO, R1, and R2 are destroyed.

TTY (%0FDC)
Receive and echo at the terminal a line of characters up to the first carriage return. The string is stored in a buffer pointed to by R2. R1 contains the size of the buffer. If the size of the string received exceeds the size of the buffer, the zero flag is set. All lower case alpha characters are converted to upper case before being stored in the buffer. R1 returns the actual number of characters received from the terminal. The contents of RO and R2 are destroyed.

CRLF (%0FD4)
Output a carriage return followed by a line feed to the terminal. RO is destroyed.

EXPANSION
Chip decoding for extra EPROM and RAM and I/O devices exists. To connect additional Z-BUS peripherals, for example, the device is wired to the Z-BUS signals required and an unused chip select line is connected to the chip select input of the peripheral. Other peripheral devices can be connected, but they may require additional circuitry in order to interface to the Z-BUS.

Additional Z6132 RAM devices can be connected directly to the Z-BUS in parallel with the existing RAMs; the only difference being the chip select lines, which should be selected from currently unused outputs. Extra EPROMs can be added in a similar manner. There is enough EPROM decoding to fill the entire 64K byte address space with 2732 EPROMs, and enough RAM decoding to do the same with Z6832 RAMs. The user can select either RAM or EPROM.

Any expansion beyond two additional peripheral chips should be accompanied by the addition of 74LS245 buffers on the Address/Data lines. Buffering is already present on AS, DS, R/W, B/W and S10-S13. If 74LS245 buffers are added, their direction should be controlled so that they drive from the CPU to the outside world except during the time that Data Strobe is active during a read operation.
Figure 1a. SSBC Schematic
Figure 1c. SS8C Schematic
Interfacing the Z8500 Peripherals to the 68000

Zilog

June 1983

INTRODUCTION

This application note discusses interfacing Zilog's Z8500 family of peripherals to the 68000 microprocessor. The Z8500 peripheral family includes the Z8536 Counter/Timer and Parallel I/O Unit (CIO), the Z8038 FIFO Input/Output Interface Unit (FIO), and the Z8530 Serial Communications Controller (SCC). This document discusses the Z8500/68000 interfaces and presents hardware examples and verification techniques. One of the three hardware examples given in this application note shows how to implement the Z8500/68000 interface using a single-chip programmable logic array (PAL).

This application note about interfacing supplements the following documents, which discuss the individual components of the interface.

- Z8036 Z-CIO/Z8536 CIO Technical Manual (document number 00-2091-01)
- Z8038 Z-FIO Technical Manual (document number 00-2051-01)
- Z8030/Z8530 SCC Technical Manual (document number 00-2057-01)
- Monolithic Memories Bipolar LSI 1982 Databook

This application note is divided into four sections. The first section gives a general description of the Z8500 family and discusses pin functions, interrupt structures, and the programming of operating modes. The second section discusses the Z8500 interface itself. It shows how the different Z8500 control signals are generated from the 68000 signals and summarizes the critical timings for the three types of bus cycle. The third section shows three examples of implementing the 68000-to-Zilog-peripheral interface. The fourth section suggests methods of verifying the interface design by checking the three different types of bus cycle: Read, Write, and Interrupt Acknowledge.

GENERAL Z8500 FAMILY DESCRIPTION

The Z8500 family is made up of programmable peripherals that can interface easily to the bus of any nonmultiplexed CPU microprocessor, such as the 68000. The three members of this family, the CIO, SCC, and FIO, can solve many design problems. The peripherals' operating modes can be programmed simply by writing to their internal registers.

Programming the Operating Modes

The CPU can access two types of register: Control and Data. Depending on the peripheral, registers are selected with either the A0, A1, A/B, or D/E function pins.

Peripheral operating modes are initialized by programming internal registers. Since these registers are not directly addressable by the CPU, a two-step procedure using the Control register is required: first, the address of the internal register is written to the Control register, then the data is written to the Control register. A state machine determines whether an address or data is being written to the Control register. Reading an internal register follows a similar two-step...
procedure: first, the address is written, then the data is read.

The Data registers that are most frequently accessed, for example, the SCC's transmit and receive buffer, can be addressed directly by the CPU with a single read or write operation. This reduces overhead in data transfers between the peripheral and CPU.

GENERATING Z8500 CONTROL SIGNALS

This section shows how to generate the Z8500 control signals. To simplify the discussion, the section is divided into two parts. The first part takes each individual Z8500 signal and shows how it is generated from the 68000 signals. The second part discusses the Z8500 timing that must be met when generating the control signals.

Z8500 Signal Generation

The right-hand side of Table 1 lists the Z8500 signals that must be generated. Each of these signals is discussed in a separate paragraph.

A0, A1, A/B, D/C. These pins are used to select the peripheral's Control and Data registers that program the different operating modes. They can be connected to the 68000 A1 and A2 Address bus lines.

CE. Each peripheral has an active Low Chip Enable that can be derived by ANDing the selected address decode and the 68000's Address Strobe (AS). The active Low AS guarantees that the 68000 addresses are valid.

D0-D7. The Z8500 Data bus can be directly connected to the lowest byte (D0-D7) of the 68000 Data bus.

IEI and IEO. The peripherals use these pins to decide the interrupt priority. The highest priority device should have its IEI tied High. Its IEO should be connected to the IEI pin of the next highest priority device. This pattern continues with the next highest priority peripheral, until the peripherals are all connected, as shown in Figure 1.

INT. The interrupt request pins for each peripheral in the daisy chain can be wire-ORed and connected to the 68000's ILPn pins. The 68000 has seven interrupt levels that can be encoded into the ILP0, ILP1, and ILP2 pins. Multiple 68000 interrupt levels can be implemented by using a multiplexer like the 74LS148.

Table 1. Z8500 and 68000 Pin Functions

<table>
<thead>
<tr>
<th>68000 Signals</th>
<th>Function</th>
<th>Z8500 Signals</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1-A23</td>
<td>Address bus</td>
<td>A0, A1, A/B, D/C*</td>
<td>Register select</td>
</tr>
<tr>
<td>AS</td>
<td>Address Strobe</td>
<td>CE</td>
<td>Chip Enable</td>
</tr>
<tr>
<td>CLK</td>
<td>68000 clock (8 MHz)</td>
<td>D0-D7</td>
<td>Data bus</td>
</tr>
<tr>
<td>D0-D15</td>
<td>Data bus</td>
<td>IEI, IEO</td>
<td>Interrupt daisy chain</td>
</tr>
<tr>
<td>DTACK</td>
<td>Data Transfer Acknowledge</td>
<td></td>
<td>control</td>
</tr>
<tr>
<td>FC0-FC2</td>
<td>Processor status</td>
<td>INT</td>
<td>Interrupt Request</td>
</tr>
<tr>
<td>ILP0-ILP2</td>
<td>Interrupt request</td>
<td>INTACK</td>
<td>Interrupt Acknowledge</td>
</tr>
<tr>
<td>R/W</td>
<td>Read/Write</td>
<td>PCLK</td>
<td>Peripheral Clock</td>
</tr>
<tr>
<td>VMA</td>
<td>Valid Memory Address</td>
<td>RD</td>
<td>Read strobe</td>
</tr>
<tr>
<td>VPA</td>
<td>Valid Peripheral Address</td>
<td>WR</td>
<td>Write strobe</td>
</tr>
</tbody>
</table>

* The register select pins on each peripheral have different names.
**INTACK.** The INTACK pin signals the peripheral that an Interrupt Acknowledge cycle is occurring. The following equation describes how INTACK is generated:

\[ \text{INTACK} = (FC_0) \cdot (FC_1) \cdot (FC_2) \cdot (AS) \]

The 68000 FC0-FC2 are status pins that indicate an Interrupt Acknowledge when they are all High. They should be ANDed with inverted AS to guarantee their validity. The INTACK signal must be synchronized with PCLK to guarantee set-up and hold times. This can be accomplished by changing the state of INTACK on the falling edge of PCLK. If the INTACK pin is not used, it must be tied High.

**PCLK.** The SCC and CIO require a clock for internal synchronization. The clock can be generated by dividing down the 68000 CLK.

**RD.** The Read strobe goes active Low under three conditions: hardware reset, normal Read cycle, and an Interrupt Acknowledge cycle. The following equation describes how RD is generated:

\[ \text{RD} = [(R/W) \cdot (AS) + \text{RESET}] \]

Forcing RD and WR simultaneously Low resets the peripherals.

**Z8500 Timing Cycles**

This section discusses the timing parameters that must be met when generating the control signals. The Z8500 family uses the control signals to communicate with the CPU via three types of bus cycle: Read, Write, and Interrupt Acknowledge.

The Read strobe timing must meet both the Read timing and Interrupt Acknowledge timing discussed in the following section. In addition to enabling the Data bus drivers, the falling edge of RD sets the Interrupt Under Service (IUS) bits during an Interrupt Acknowledge cycle.

**WR.** This signal strobes data into the peripheral. A data-to-write setup time requires that data be valid before WR goes active Low. The equation for generating the WR strobe is made up of two components: an active reset and a normal Write cycle, as shown in the following equation:

\[ \text{WR} = [(R/W) \cdot (AS) + \text{RESET}] \]

Forcing RD and WR simultaneously Low resets the peripherals.

**Figure 1. Peripheral Interrupt Daisy Chain**

...
The discussion that follows pertains to the 4 MHz peripherals, but the 6 MHz devices have similar timing considerations.

Although the peripherals have a standard CPU interface, some of their particular timing requirements vary. The worst-case parameters are shown below; the timing can be optimized if only one or two of the Z8500 family devices are used.

Read Cycle

The Read cycle transfers data from the peripheral to the CPU. It begins by selecting the peripheral and appropriate register (Data or Control). The data is gated onto the bus with the RD line. A setup time of 80 ns from the time the register select inputs (A/B, C/D, A0, A1) are stable to the falling edge of RD guarantees that the proper register is accessed. The access time specification is usually measured from the falling edge of RD to valid data and varies between peripherals. The SCC specifies an additional register select to valid data time. The Read cycle timing is shown in Figure 2.

Write Cycle

The Write cycle transfers data from the CPU to the peripheral. It begins by selecting the peripheral and addressing the desired register. A setup time of 80 ns from register select stable to the falling edge of WR is required. The data must be valid prior to the falling edge of WR. The WR pulse width is specified at 400 ns. Write cycle timing is shown in Figure 2.

Interrupt Acknowledge Cycle

The Z8500 peripheral interrupt structure offers the designer many options. In the simplest case, the Z8500 peripherals can be polled with interrupts disabled. If using interrupts, the timing shown in Figure 2 should be observed. (Detailed discussions of the interrupt processing can be found in the Zilog Data Book, document number 00-2034-02.) An interrupt sequence begins with an INT going active because of an interrupt condition. The CPU acknowledges the interrupt with an INTACK signal.

---

Figure 2. Z8500 Interface Timing (4 MHz)
A daisy-chain settle time (dependent upon the number of devices in the chain) ensures that the interrupts are prioritized. The falling edge of RD causes the IUS bit to be set and enables a vector to go out on the bus.

The table given in Figure 1 can be used to calculate the amount of settling time required by a daisy chain. Even if there is only one peripheral in the chain, a minimum settling time is still required because of the internal daisy chain. The first column specifies the amount of settling time for only one peripheral. If there are two peripherals, the time is computed by adding together the times shown in the first and the last columns. For each additional peripheral in the chain, the time specified in the middle column is added.

**Recovery Time**

The read/write recovery time specifies a minimum amount of time between Read or Write cycles to the same peripheral. The recovery time differs among peripherals and is summarized in Figure 3. In most cases, this parameter is met because of the time required for instruction fetches. The recovery time specification does not have to be met if CE is deselected when Read or Write occurs.

**68000 INTERFACE EXAMPLES**

This section shows three examples, presented in increasing order of complexity, for interfacing Zilog's 4 MHz Z8500 peripherals to an 8 MHz 68000. Faster CPUs or peripherals can be used by modifying some of the timing. These examples suggest possible ways of implementing the interface but may require some modifications to operate properly. They were chosen because they give the user a variety of interface design ideas. The first example uses a minimum amount of TTL logic to implement the interface because the Valid Peripheral Address (VPA) cycle meets the Z8500 timing requirements. In this mode the 68000 accepts only nonvectored interrupts. The second example uses the Data Transfer Acknowledge (DTACK) pin. This interface allows faster operation and makes use of the Z8500's 8-bit vectored interrupts. The third example also uses a DTACK cycle and is similar to the second, except the external logic is integrated into a single chip, the PAL20X10 programmable array logic.

**EXAMPLE 1: A TTL Interface Using a VPA Cycle**

The 68000 has a special input pin, Valid Peripheral Address (VPA), that can be activated by the Z8500 chip select logic at the beginning of the cycle to indicate to the 68000 that a peripheral is being accessed. This generates a special Read/Write cycle that meets the peripheral timing requirements. This cycle allows the Z8500 control signals to be generated easily. The 68000 responds to interrupts using an autovector and the Z8500 can be programmed not to return a vector.

![Figure 3. Recovery Time](image)

**NOTE.** The diagram shows that the recovery time is measured between consecutive reads and writes only if the peripheral is selected.
Figure 4 shows how the hardware can be implemented. PCLK is generated by dividing down the 68000 CLK. RD, WR, and INTACK are simply ANDed 68000 signals. The worst-case daisy-chain settle time is 450 ns. Connecting INT to IPL0 generates a level 1 interrupt. The internal registers are accessed by A0, A1, D/C, and A/B, which can be the 68000 lowest order addresses. The timing is shown in Figure 5.

![Figure 4. Interface Using the VPA Cycle](image)

![Figure 5. VPA Cycle Timing](image)
Functional Description

VPA is pulled Low at the beginning of the cycle and the CPU automatically inserts Wait states until E is synchronized.

\[ \text{VPA} = [(\text{AS})'(\text{CE})] \]

\[ \text{RD} = [(\text{CE})'(\text{VMA})'(\text{R/W})] \]

\[ \text{WR} = [(\text{CE})'(\text{VMA})'(\text{R/W})] \]

\[ \text{INTACK} = [(\text{FCO})'(\text{FC1})'(\text{FC2})'(\text{AS})] \]

**EXAMPLE 2: A TTL Interface Using DTACK Cycles**

Using the 68000 Data Transfer Acknowledge (DTACK) cycle is a second way of interfacing to the Z8500 peripherals. The 68000 inserts Wait states until the DTACK input is strobed Low to complete the transfer. In addition to generating the control signals, the interface logic must also generate DTACK.

The timing shown in Figure 6 can be generated by the hardware shown in Figure 7. The 8-bit Shift register (74LS164) is used to generate the proper timing. At the beginning of each cycle, QA (Figure 7) is set High for one PCLK cycle and then reset. This pulse is shifted through the QA-QH outputs and is used to generate RD, WR, and DTACK signals. Some of the extra Wait states can be eliminated by tapping the Shift register sooner (e.g., QC).

**EXAMPLE 3: Single-Chip Pal Interface**

This example illustrates how to interface the 4 MHz Z8500 peripherals to the 8 MHz 68000 using a PAL20X10 device to generate all the required control signals. The PAL reduces the required interface logic to a single chip, thus minimizing board space. This interface offers flexibility because the internal logic can be reprogrammed without changing the pin functions. The PAL uses 68000 signals to generate Read, Write, and Interrupt Acknowledge cycles. In addition to generating the Z8500 control signals, the PAL also generates a DTACK to inform the 68000 of a completed data transfer cycle. This allows the 68000 to use the peripheral's vectored interrupts.

![Figure 6. Timing for DTACK Interface](image)
Figure 7. Hardware Diagram for DTACK Interface

Functional Description

Figure 8 shows the PAL's pin functions. The PAL generates five control signals, of which four (WR, RD, C0, and INTACK) go to the Z8500 and one (DTACK) goes to the 68000. The remaining signals are used internally to generate these outputs. Timing diagrams for the Read, Write, and Interrupt Acknowledge cycles are shown in Figure 9.

The PAL uses a 4-bit downcounter to generate the proper placement of the control signals where C0 is the least-significant bit and C3 is the most-significant bit.

CLC 1 24 VCC
CS 2 23 ACK
OE 3 22 WR
TEST 4 21 RD
AS 5 20 DTACK
R/W 6 19 NC
FC0 7 18 CYC
FC1 8 17 CS
FC2 9 16 ET
RESET 10 15 CE
NO 11 14 CE
GND 12 13 CE

Figure 8. PAL Pinout
most-significant bit. All of the PAL is clocked with the rising edge of the 68000's CLK. The counter toggles between counts 14 and 15 and starts counting down when AS goes active. The counter goes back to toggling when AS goes inactive. CYC goes active low at the same time the counter starts counting down. The equations in Figure 10 can be entered into a development board to program the PAL.

---

**Figure 9. PAL Interface Timing**
PAL20X10
P7089 (10)
MC68000 TO ZILOG PERIPHERAL INTERFACE
MMI, SUNNYVALE, CA
CLK /CS NC TEST /AS RW
FC2 FC1 FCO /RESET NC GND
/OE /C3 /C2 /C1 /CO /CYC
NC /DTK /RD /WR /ACK VCC

\[
\begin{align*}
C_0 & := /CO*/TEST & \text{; COUNT/HOLD (LSB)} \\
C_1 & := /RESET*AS*C1 & \text{; HOLD} \\
& \quad +: /RESET*AS*CO & \text{; DECREMENT} \\
C_2 & := /RESET*AS*C2 & \text{; HOLD} \\
& \quad +: /RESET*AS*CO*C1 & \text{; DECREMENT} \\
C_3 & := /RESET*AS*C3 & \text{; HOLD} \\
& \quad +: /RESET*AS*CO*C1*C2 & \text{; DECREMENT} \\
DTK & := /RESET*/ACK*CYC*C3*/C2*/C1*/CO*CS \\
& \quad + /RESET*ACK*CYC*C3*/C2* C1*/CO & \text{; DTACK FOR RD/WR CYCLE} \\
& \quad + /RESET*ACK*CYC*C3*/C2* CO & \text{; DTACK FOR INTERRUPT} \\
& \quad + /RESET*ACK*CYC & \text{; OPERATION} \\
CYC & := /RESET*AS*/CYC*CO & \text{; NEW CYCLE STARTED} \\
& \quad + /RESET*AS* CYC & \text{; PROCESSING OF CYCLE} \\
& \quad + /RESET*CYC*DTK & \text{; END OF CYCLE} \\
RD & := /RESET*CYC*/ACK*/RW* C3*/C2*CS \\
& \quad + /RESET*CYC*/ACK*/RW*/C3*C2*C1*CO*CS & \text{; NORMAL READ OPERATION} \\
& \quad + /RESET*CYC* ACK*RW* C3 & \text{; NORMAL READ OPERATION} \\
& \quad + /RESET*CYC & \text{; READ DURING OPERATION} \\
WR & := /RESET*CYC*/ACK*/RW* C3*/C2*CS \\
& \quad + /RESET*CYC*/ACK*/RW*/C3* C2*C1*CO*CS & \text{; WRITE} \\
& \quad + /RESET*CYC & \text{; WRITE} \\
& \quad + /RESET*FC0*FC1*FC2*AS* CYC*/CO & \text{; INTERRUPT ACKNOWLEDGE} \\
& \quad + /RESET*FC0*FC1*FC2*CYC & \text{; INTERRUPT ACKNOWLEDGE}
\end{align*}
\]

Figure 10. PAL Equations

Hardware Diagram

The hardware diagram of the PAL interface is shown in Figure 11. The 68000 signals CLK, CS, AS, R/W, FC0, FC1, and FC2 are used to generate the Z8500 control signals. The control signals are synchronous with the rising edge of the 68000's CLK. TEST and OE must be grounded. CO is used to enable DTACK, RD, and WR as shown in the equations. The Z8500 INT is connected to ILP0, which generates a 68000 level 1 interrupt. The peripherals are memory-mapped into the highest 64K byte block of memory, where A17-A23 equals "FFH". Addresses A4-A6 are used to select the peripheral; A1-A3 select the internal registers. Table 2 shows the peripheral's memory map.
Figure 11. PAL Hardware Diagram
Table 2. Peripheral Memory Map

<table>
<thead>
<tr>
<th>Peripheral</th>
<th>Register</th>
<th>Hex Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCC (Z8530)</td>
<td>Channel B Control</td>
<td>FF0020</td>
</tr>
<tr>
<td></td>
<td>Channel B Data</td>
<td>FF0022</td>
</tr>
<tr>
<td></td>
<td>Channel A Control</td>
<td>FF0024</td>
</tr>
<tr>
<td></td>
<td>Channel B Data</td>
<td>FF0026</td>
</tr>
<tr>
<td>CIO (Z8536)</td>
<td>Port C's Data Register</td>
<td>FF0010</td>
</tr>
<tr>
<td></td>
<td>Port B's Data Register</td>
<td>FF0012</td>
</tr>
<tr>
<td></td>
<td>Port A's Data Register</td>
<td>FF0014</td>
</tr>
<tr>
<td></td>
<td>Control Register</td>
<td>FF0016</td>
</tr>
<tr>
<td>FIO (Z8038)</td>
<td>Data Registers</td>
<td>FF0000</td>
</tr>
<tr>
<td></td>
<td>Control Registers</td>
<td>FF0002</td>
</tr>
</tbody>
</table>

INTERFACE VERIFICATION TECHNIQUES

This section suggests possible ways of verifying the Read, Write, and Interrupt Acknowledge cycles.

Read Cycle Verification

The Read cycle should be checked first because it is the simplest operation. The Z8500 should be hardware reset by simultaneously pulling RD and WR Low. When the peripheral is in the reset state, the Control register containing the reset bit can be read without writing the pointer. Reading back the FIO or CIO Control register should yield a 01H.

The SCC's Read cycle can be verified by reading the bits in RR0. Bits D2 and D6 are set to 1 and bits D0, D1, and D7 are 0. Bits D3-D5 reflect the input pins DCD, SYNC, and CTS, respectively.

Write Cycle Verification

The Write cycle can be checked by writing to a register and reading back the results. Both the CIO and FIO must have their reset bits cleared by writing 00H to their Control registers and reading back the result. The SCC can be checked by writing and reading to an arbitrary read/write register, for example, the Time Constant register (WR12 or WR13).

Interrupt Acknowledge Cycle Verification

Verifying an Interrupt Acknowledge (INTACK) cycle consists of several steps. First, the peripheral makes an Interrupt Request (INT) to the CPU. When the processor is ready to service the interrupt, it initiates an Interrupt Acknowledge (INTACK) cycle. The peripheral then puts an 8-bit vector on the bus, and the 68000 uses that vector to get to the correct service routine. This test checks the simplest case.

First, load the Interrupt Vector register with a vector, disable the Vector Includes Status (VIS), and enable interrupts (IE = 1, MIE = 1, IEI = 1). Disabling VIS guarantees that only one vector is put on the bus. The address of the service routine corresponding to the 8-bit vector number must be loaded into the 68000's vector table.

Initiating an interrupt sequence in the FIO and CIO can be accomplished by setting one of the interrupt pending (IP) bits and seeing if the 68000 jumps to the service routine (setting a breakpoint at the beginning of the service routine is an easy way to check if this has happened).

Initiating an interrupt sequence in the SCC is not quite as simple because the IP bits are not as accessible to the user. An interrupt can be generated indirectly via the CTS pin by enabling the following: CTS IE (WR15 20), EXT INT EN (WR1 01), and MIE (WR9 0B). Any transition on the CTS pin can initiate the interrupt sequence. The interrupt can be re-enabled by RESET EXT/STATUS INI (WRO 10) and RESET HIGHEST IUS (WRO 38).

CONCLUSION

Zilog's Z8500 family of nonmultiplexed Address/Data bus peripherals can interface easily with the 68000 and provide all the support required in a high-performance microprocessor system. The many features offered by the SCC, FIO, and CIO solve many system design problems by making interfacing to the external world easy. These intelligent peripherals also greatly enhance the system performance by relieving the CPU of many burdensome overhead tasks. Additionally, the powerful interrupt structure allows the 68000 to use vectors and reduce interrupt response time.
Microcomputer systems based on Intel's 8086 and 8088 CPUs can take advantage of the advanced features of Zilog's Z8000 series of microprocessor peripherals with a minimal amount of external logic. These devices are easily integrated and can satisfy many of the peripheral support requirements in a typical 8086/8088-based system. This Application Note discusses a general design that enables the 8086/8088 to interface with Zilog's Serial Communications Controller (Z8030 Z-SCC), Counter/Timer - Parallel I/O Unit (Z8036 Z-CIO), and FIFO I/O Controller (Z8038 Z-FIO). Discussions of the Z8500 peripherals (non-multiplexed address and data bus versions) can be found in other Zilog documents.

BUS INTERFACE

The Z8000 peripherals (also called Z-BUS peripherals) lend themselves conveniently to 8086/8088-based designs because of the multiplexed address/data bus architecture. There is no need for an external address latch because the Z8000 peripherals latch addresses internally at the beginning of each bus cycle. Furthermore, the peripherals allow the CPU direct access to all of their data and control registers. Figure 1 shows the interface logic that translates the signals generated by the 8086/8088 into the necessary Z-BUS signals, and Table 1 gives a description of each signal.

Note.
1. The source of PCLK can, but need not, be derived from the System CLK.
2. Does not apply to Z-FIO.
3. AD9-AD16 on 8088.
4. IOM on 8088.

Figure 1. Interface Logic
### Table 1. Signal Descriptions

#### 8086/8088 Signals

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MN/MX</td>
<td>Minimum/Maximum. This input is pulled high so that the CPU will operate in the &quot;Minimum Mode.&quot;</td>
</tr>
<tr>
<td>DT/R</td>
<td>Data Transmit/Receive. DT/R is high on write operations and low on read operations.</td>
</tr>
<tr>
<td>ALE</td>
<td>Address Latch Enable. ALE is used to latch addresses during the first T state of each bus cycle so that the bus can then be free to transfer data.</td>
</tr>
<tr>
<td>RD</td>
<td>Read. RD strobes data into the CPU on read operations.</td>
</tr>
<tr>
<td>WR</td>
<td>Write. WR strobes data out of the CPU on write operations.</td>
</tr>
<tr>
<td>AD0-AD15</td>
<td>This is the 16-bit, multiplexed address/data bus on the 8086. The 8088 has a low order address/data bus, AD0-AD7, and a high order address bus, A0-A15.</td>
</tr>
<tr>
<td>M/IO</td>
<td>Memory/Input-Output. This output distinguishes between memory and I/O accesses. On the 8086 it is high on memory accesses and low on I/O accesses. On the 8088, the polarity is reversed (IO/W).</td>
</tr>
</tbody>
</table>

#### Z-BUS Signals

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/ W</td>
<td>Read/Write. This input tells the peripheral whether the present access is a read or write. It is generated by inverting DT/R of the 8086/8088.</td>
</tr>
<tr>
<td>AS*</td>
<td>Address Strobe. AS is the main clock signal for the Z-BUS peripherals. It is used to initiate bus cycles by latching the address along with CS0 and INTACK. It is generated by inverting ALE of the 8086/8088.</td>
</tr>
<tr>
<td>DS*</td>
<td>Data Strobe. When the Z-BUS peripheral is selected, DS gates data onto or from the bus, depending on the state of R/ W. It is generated from the 8086/8088 signals RD and WR as shown in Figure 1.</td>
</tr>
<tr>
<td>INTACK</td>
<td>Interrupt Acknowledge. When low, this signal tells the peripheral that the present cycle is an Interrupt Acknowledge cycle.</td>
</tr>
<tr>
<td>AD0-AD7</td>
<td>Address/Data Bus. This bus is connected directly to AD0-AD7 of the 8086/8088. It is possible to connect it to AD0-AD15 of the 8086 as long as the 8086 doesn't expect to read an interrupt vector from the peripheral during interrupt acknowledge transactions.</td>
</tr>
<tr>
<td>CS0,CS1</td>
<td>Chip selects. CS0 is active low and is latched with the rising edge of AS. CS1 is active high and is unlatched. In this interface, CS1 is pulled high while CS0 is generated from the address decode logic.</td>
</tr>
<tr>
<td>PCLK</td>
<td>Peripheral Clock. This signal does not apply to the Z-FIO. It can also be omitted from the Z-CIO interface if the chip is not used as a timer, its REQUEST/WAIT logic is disabled, and it does not employ deskew timers in its handshake operations. The maximum frequency of PCLK is 4 or 6 MHz, depending on the grade of the component, and it can be asynchronous to the system clock.</td>
</tr>
</tbody>
</table>

*A hardware reset of a Z-BUS peripheral is performed by driving AS and DS low simultaneously.*
BUS TIMING

Each 8086/8088 bus cycle begins with an ALE pulse, which is inverted to become Address Strobe (AS). The trailing edge of this strobe latches the register address, as well as the states of CS0 and INTACK within the peripheral. DS is then used to gate data into (write) or from (read) the selected register, provided that an active CS0 has been latched. To assure proper timing, the AC Characteristics of both the 8086/8088 and the Z-BUS peripherals, must be examined. The paragraphs that follow discuss all of the significant timing considerations that pertain to Read/Write operations in this interface.

ADDRESS AND CHIP SELECT (CS0) SETUP TIMES. The 4 MHz Z-BUS peripherals require that the stable address setup time prior to AS be at least 30 ns. Since the 5 MHz 8086/8088 is guaranteed to provide valid addresses at least 60 ns before Address Latch Enable (ALE) goes low, this requirement is easily satisfied. The CS0 setup time is of no concern because the Z8000 peripherals require no CS0 setup time prior to AS.

ADDRESS AND CHIP SELECT (CS0) HOLD TIMES. The Z-BUS specifications require that the address and CS0 remain valid a certain period of time after the rising edge of AS. These minimum values are 50 and 60 ns respectively for the 4 MHz devices. At 5 MHz, the 8086/8088 will hold its address at least 60 ns after ALE goes inactive. Although this is equal to the minimum CS0 hold time, a safe margin will be maintained if the propagation delay between the address going invalid to CS0 rising exceeds the propagation delay between ALE falling and AS rising.

ADDRESS STROBE (AS) TO DATA STROBE (DS) DELAY. The 4 MHz peripherals need a 60 ns delay between AS rising and DS falling. This parameter is of no concern on write cycles because the D-flop will delay DS until the beginning of T3 (See Figure 2). On read cycles, DS follows RD, so the delay between AS and DS is approximately equal to the delay between ALE and RD. If ALE falls at its latest possible point in time and RD falls at its earliest point, the time between these two edges would be about 60 ns. This result is unrealistic, however, because a delay in the termination of ALE
**will always lead to a delay in the activation of RD. The actual time between the two edges is well over 100 ns.**

**ADDRESS SETUP TIME TO DATA STROBE (DS).** The 4 MHz Z-CIO and Z-FIO require that the stable address setup time to DS be at least 130 ns. Since the delay between AS rising and DS falling is well over 100 ns, and since the address setup time to AS is at least 60 ns, this requirement is easily satisfied.

**DATA STROBE (DS) LOW WIDTH.** The minimum Data Strobe Low Width of the 4 MHz Z-BUS peripherals is 390 ns. On read cycles, DS will have the same width as RD, which is at least 325 + 200Nw ns, where Nw is the number of wait states in the bus cycle. On write cycles, the D-flip-flop will shorten this minimum width to 210 + Nw 200 ns. One wait state (Tw) in the bus cycle will ensure a sufficiently wide Data Strobe for both types of bus cycles. A discussion of wait state generation is presented in the next section.

**WRITE DATA SETUP AND HOLD TIMES.** On write cycles, the Z-BUS peripherals require the CPU to put valid data on the bus at least 30 ns before DS goes active, and to hold it there at least 30 ns after DS terminates. D-flip-flop in Figure 2 guarantees the setup time by delaying the falling edge of WR until the next falling edge of SYS CLK (Figure 2.). The Hold Time is also guaranteed because the 8086/8088 will hold valid data at least 90 ns after the termination of WR.

**READ DATA SETUP AND HOLD TIMES.** When the 8086/8088 reads from memory or peripherals, it requires them to put valid data on the bus at least 30 ns before DS goes active, and to hold it there at least 30 ns after DS terminates. Since the Z-BUS peripherals will provide valid data early in Tw and will hold it until after DS terminates, these parameters are well within the specifications.

**VALID ACCESS RECOVERY TIME.** This parameter refers to the time between consecutive accesses to a given peripheral. If the 4 MHz Z-SCC is accessed twice, then the time between DS rising in the first access and DS falling in the second access, must be at least 6 PCLK cycles plus 200 ns (i.e. 1700 ns for a 4 MHz PCLK). The Valid Access Recovery Time for the 4 MHz Z-CIO and Z-FIO is 1000 ns, and this can't possibly be violated with a 5 MHz 8086/8088 since there will always be at least one instruction fetch cycle in between I/O accesses, and 1000 ns translates into only 5 clock cycles at 5 MHz.

**WAIT STATE GENERATION**

The previous section explained why the 4 MHz Z8000 peripherals need to place a wait state in I/O bus cycles when interfaced to the 5 MHz 8086/8088. The following two examples illustrate how wait state generation can be implemented. Since 8086/8088 - based systems typically use an 8284 Clock Chip, which synchronizes the CPU's READY input with the system clock, the task reduces to designing a circuit that will control the RDY1 input of the 8284 (RDY2 is assumed to be grounded).

**SINGLE WAIT STATE GENERATION.** For the processor to enter a wait state after T3, the RDY1 input must be low during the falling edge of SYS CLK at the end of T2. Then, for the processor to enter T4 after the wait state, RDY1 must be high during the next falling edge of SYS CLK. To make sure that these levels are well-established during their sampling windows, the single wait state generator should toggle RDY1, using the clock edges that precede the sampling edges (Figure 4). The circuit in Figure 5 performs this function and generates a single wait state when one of the CSq inputs is active.
MULTIPLE WAIT STATE GENERATION. Though Read/Write operations require only one wait state, Interrupt Acknowledge transactions need multiple wait states to allow for daisy-chain settling, which is explained in the next section. The following discussion introduces a multiple wait state generator and serves as a basis for understanding the subsequent Interrupt Acknowledge Circuit.

In the preceding discussion of the single wait state generator, we established that RDY1 must be high at the end of T3 for the processor to enter T4 after the wait state. In general, the 8086/8088 will continue to insert wait states until RDY1 is driven high. In fact, the number of wait states will be equal to the number of clock cycles that RDY1 is held low after the rising clock edge in T2.

A convenient way to implement a multiple wait state generator is to use a serial shift register such as a 74LS164. Figure 6 shows a wait state generator that requests one wait state on Read/Write cycles, and up to seven wait states on Interrupt Acknowledge cycles. When RD, WR, or INTA goes active, the 74LS164 is taken out of the clear state and logic "ones" are allowed to shift sequentially from QA to QH. On Read/Write cycles, RDY1 is held low until the leading "one" appears at QB, and on Interrupt Acknowledge cycles, RDY1 is held low until the leading "one" appears at QH. The next section shows how INTACK can be generated and discusses the complete interrupt interface.

INTERRUPTS

In Figure 1 the INTACK input to the Z-BUS peripherals is pulled high. This does not mean that the peripheral can't interrupt the CPU; it just means that it won't respond to the CPU's interrupt acknowledge. The designer can, however, implement a circuit that will drive INTACK, and allow the 8086/8088 to properly acknowledge the interrupts of the Z-BUS peripherals. This section examines the interrupt acknowledge protocols of the Z-BUS peripherals and the 8086/8088, then proceeds to show how they can be made compatible.

Z-BUS INTERRUPT ACKNOWLEDGE PROTOCOL. The Z-BUS peripherals typically use the daisy-chain technique of priority interrupt control. In this scheme the peripherals are connected together via an interrupt daisy chain formed with their IEI (Interrupt Enable Input) and IEO (Interrupt Enable Output) pins (Figure 7). The interrupt sources within a device are similarly chained together, with the overall effect being a daisy chain connecting all of the interrupt sources. The daisy chain allows higher priority interrupt sources to preempt lower priority sources and, in the case of simultaneous interrupt requests, determines which request will be acknowledged.

In each bus cycle the Z-BUS peripherals use the rising edge of AS to latch the state of INTACK. If a low INTACK is latched, then the present cycle is an Interrupt Acknowledge cycle and the daisy chain determines which interrupt source is being acknowledged in the following way. Any interrupt source that has an interrupt pending and is not masked from the chain will hold its IEO low.

![Figure 6. Multiple Wait State Generator](image)

![Figure 7. A Z-BUS Interrupt Daisy Chain](image)
Similarly, sources that are currently under service (i.e. have their IUS bit set) will also hold their IEO lines low. All other interrupt sources make IEO follow IEl. The result is that only the highest priority, unmasked source with an interrupt pending will have a high IEl input; only this peripheral will be allowed to transfer its vector to the system bus when the Data Strobe is issued during the Interrupt Acknowledge cycle.

To make sure that the daisy chain has settled by the time DS gates the vector onto the bus, the Z-BUS peripherals require a sufficient delay between the rising edge of AS and the falling edge of DS in INTACK cycles. The amount of delay required can be calculated using Table 2. For a particular daisy chain, the minimum delay is: Thigh for the highest priority device, plus Tlow for the lowest priority device, plus Tmid for each device in between.

Table 2. Daisy Chain Settling Times for the Z-BUS Peripherals (in ns)

<table>
<thead>
<tr>
<th></th>
<th>4MHz</th>
<th>6MHz</th>
<th>4MHz</th>
<th>6MHz</th>
<th>4MHz</th>
<th>6MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z-SCC</td>
<td>250</td>
<td>250</td>
<td>120</td>
<td>100</td>
<td>120</td>
<td>100</td>
</tr>
<tr>
<td>Z-C10</td>
<td>350</td>
<td>250</td>
<td>150</td>
<td>100</td>
<td>100</td>
<td>70</td>
</tr>
<tr>
<td>Z-F10</td>
<td>350</td>
<td>250</td>
<td>150</td>
<td>100</td>
<td>100</td>
<td>70</td>
</tr>
</tbody>
</table>

8086/8088 INTERRUPT ACKNOWLEDGE PROTOCOL. If the 8086/8088 receives an interrupt request (via its INTR pin) while its Interrupt Flag is set, then it will execute an Interrupt Acknowledge sequence. The sequence consists of two identical INTA bus cycles with two idle clock cycles in between (Figure 8). In both bus cycles, RD and WR remain inactive while an INTA strobe is issued with the same timing as a WR strobe. The 8086/8088 requires an interrupt vector to appear on AD0 - AD7 at least 30 ns before the beginning of T4 in the second INTA cycle. This protocol is normally used to read vectors from the 8259A Interrupt Controller but it can easily be adapted to the Z-BUS Interrupt Acknowledge Protocol, as illustrated in the following paragraphs.

**INTERRUPT ACKNOWLEDGE COMPATIBILITY.** The first function of the Interrupt Acknowledge circuit, shown in figure 9, is to generate the Z-BUS INTACK signal using INTA from the 8086/8088. Since INTA goes active after ALE has terminated, the peripherals will not latch an active INTACK during the first INTA cycle. However, if the rising edge of INTA is used to toggle INTACK, then an active INTACK latches with the rising edge of AS in the second INTA cycle. Thus a rising-edge triggered toggle flip-flop, as configured in Figure 9, can be used to generate INTACK. Figure 10 shows the timing relationship between INTA and INTACK.

The next function of the Interrupt Acknowledge circuit can be broken down into three operations: first, it must cause the CPU to enter a series of wait states after T3 in the second INTA cycle; then, it must activate DS after a sufficient daisy chain settling time; lastly, it must bring the CPU out of the wait state condition when the vector is available on the bus.
Figure 9 shows how the multiple wait state generator, discussed in the previous section, can be used to perform each of these operations.

While INTACK is high the circuit operates normally; the number of wait states it requests is determined by the positioning of the jumper on the Q outputs. When INTACK goes low, it operates as follows: the next activation of INTA brings the shift register out of the clear state, and logic "ones" shift into QA until they fill the entire register. When the leading "one" appears at QG, DS is driven low; when it appears at QH, the CPU is taken out of the wait state condition.

This arrangement takes advantage of the full length of the shift register and provides a daisy-chain settling time of more than 1300 ns, which allows the implementation of a chain with as many as seven Z-BUS devices. Figure 10 shows the timing of the important signals in the Interrupt Acknowledge transaction.

**HARDWARE RESET**

The designer may want to incorporate a hardware reset in the interface design. This can be accomplished with two NOR gates as shown in Figure 11. The NOR gates allow the system RESET signal to pull AS and DS low simultaneously, and hence put the peripheral in a reset state. A hardware reset is not necessary, however, because all of the peripherals are equipped with software reset commands.

![Interrupt Acknowledge Circuit](image)

**Figure 9. Interrupt Acknowledge Circuit**

![Interrupt Acknowledge Timing](image)

**Figure 10. Interrupt Acknowledge Timing**
The Z-SCC, Z-CIO, and Z-FIO can easily be designed into 8086/8088-based systems. Their data and control registers can be mapped directly into the I/O address space, and the Z-BUS control signals can be generated with a minimal amount of external logic. The user can also take advantage of the devices' interrupt control capabilities because a simple interface circuit makes their interrupt structure compatible with that of the 8086/8088.

Figure 11. Hardware Reset
INTRODUCTION

Direct Memory Access (DMA) is a data transfer method that uses special hardware to transfer data between system memory and the outside world (e.g., a peripheral I/O device) without the intervention of a Central Processing Unit (CPU).

A transfer controller usually handles all aspects of a data transfer: it provides read or write control signals and addresses to the system, updates the addresses, counts the number of words or bytes in the transfer, and signals the end of an operation. The advantage of DMA is speed. Transfers can proceed at the memory's maximum speed rather than waiting for the CPU to fetch and decode the instructions, move the data, update the addresses, and count the words or bytes. The DMA controller performs these tasks at hardware speed and reduces CPU overhead costs.

The Z8016 DMA Transfer Controller (DTC) is a high-performance 16-bit peripheral interface device designed for Z8000 processor systems. Each of the DTC's two channels can perform the following kinds of transfer: memory-to-peripheral, memory-to-memory, peripheral-to-memory, and peripheral-to-peripheral. For all DMA operations (i.e., Transfer, Search, and Transfer-and-Search), the DTC operates with either word or byte data sizes and provides a packing/unpacking capability. To eliminate the overhead needed to load the internal registers, the DTC provides an auto-chaining operation to load and reload the 13 channel registers (Figure 1b). The CPU need only load the address of the control parameter table into the

**Figure 1a. Z8016 DTC Block Diagram**

February 1983
Chain Address register and issue a Start Chain command to load the control parameters from memory into the channel's control registers.

The DTC is Z-BUS compatible and operates within the Z8000 daisy-chain, vectored-priority interrupt scheme. Additionally, a demand interleave operation is supported, which allows the DTC to surrender the system bus to the external system or to alternate between internal channels. This capability allows for parallel operations between the two channels or between a DTC channel and the CPU.

INTERFACING

A block diagram of the Z8016 DTC (Figure 1) shows the internal configuration. The internal registers are defined in Figures 2 and 3 and listed in Table 1. Figure 4 shows the interface signals. All of the input and output signals (except the clock input) are directly TTL compatible. All outputs source at least 250 µA at 2.4 V and sink up to 3.2 mA at 0.4 V.

![Diagram of Z8016 DTC Block Diagram, Channel Registers](image-url)
Figure 2. Z8016 DTC Internal Registers
DATA OPERATION FIELD

<table>
<thead>
<tr>
<th>Code/Operation</th>
<th>Operand Size</th>
<th>Transaction Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transfer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>Byte</td>
<td>Flowthrough</td>
</tr>
<tr>
<td>100X</td>
<td>Byte</td>
<td>Flowthrough</td>
</tr>
<tr>
<td>0000</td>
<td>Word</td>
<td>Flowthrough</td>
</tr>
<tr>
<td>0011</td>
<td>Byte</td>
<td>Flyby</td>
</tr>
<tr>
<td>0010</td>
<td>Word</td>
<td>Flyby</td>
</tr>
<tr>
<td>Transfer-and-Search</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>Byte</td>
<td>Flowthrough</td>
</tr>
<tr>
<td>110X</td>
<td>Byte</td>
<td>Flowthrough</td>
</tr>
<tr>
<td>0100</td>
<td>Word</td>
<td>Flowthrough</td>
</tr>
<tr>
<td>0111</td>
<td>Byte</td>
<td>Flyby</td>
</tr>
<tr>
<td>0110</td>
<td>Word</td>
<td>Flyby</td>
</tr>
<tr>
<td>Search</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>Byte</td>
<td>N/A</td>
</tr>
<tr>
<td>1110</td>
<td>Word</td>
<td>N/A</td>
</tr>
<tr>
<td>101X</td>
<td>Illegal</td>
<td></td>
</tr>
</tbody>
</table>

TRANSFER TYPE FIELD AND MATCH CONTROL FIELD

<table>
<thead>
<tr>
<th>Code</th>
<th>Transfer Type</th>
<th>Match Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Single Transfer</td>
<td>Stop on No Match</td>
</tr>
<tr>
<td>01</td>
<td>Demand Dedicated/Bus Hold</td>
<td>Stop on No Match</td>
</tr>
<tr>
<td>10</td>
<td>Demand Dedicated/Bus Release</td>
<td>Stop on Word Match</td>
</tr>
<tr>
<td>11</td>
<td>Demand Interleave</td>
<td>Stop on Byte Match</td>
</tr>
</tbody>
</table>

Figure 3. Z8016 DTC Channel Mode Register
Table 1. Z8016 DTC Internal Registers

<table>
<thead>
<tr>
<th>Device Registers</th>
<th>Port Address (Hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Register</strong></td>
<td><strong>Channel 1</strong></td>
</tr>
<tr>
<td><strong>Master Mode register</strong></td>
<td>38</td>
</tr>
<tr>
<td><strong>Command register</strong></td>
<td>2C</td>
</tr>
<tr>
<td><strong>Chain Control register</strong></td>
<td>--</td>
</tr>
<tr>
<td><strong>Temporary register</strong></td>
<td>--</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Channel Registers</th>
<th>Segment/Tag</th>
<th>Offset</th>
<th>Segment/Tag</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Address registers, chainable</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Current Address - A</td>
<td>9</td>
<td>1A</td>
<td>0A</td>
<td>18</td>
</tr>
<tr>
<td>Current Address - B</td>
<td>8</td>
<td>12</td>
<td>02</td>
<td>10</td>
</tr>
<tr>
<td>Base Address - A</td>
<td>6</td>
<td>1E</td>
<td>0E</td>
<td>C</td>
</tr>
<tr>
<td>Base Address - B</td>
<td>5</td>
<td>16</td>
<td>06</td>
<td>14</td>
</tr>
<tr>
<td>Chain Address</td>
<td>0</td>
<td>26</td>
<td>22</td>
<td>24</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Control registers, chainable</strong></th>
<th>Segment/Tag</th>
<th>Offset</th>
<th>Segment/Tag</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current Op-Count</td>
<td>7</td>
<td>32</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>Base Op-Count</td>
<td>4</td>
<td>36</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>Channel Mode* - High</td>
<td>1</td>
<td>56</td>
<td>54</td>
<td></td>
</tr>
<tr>
<td>Channel Mode* - Low</td>
<td>1</td>
<td>52</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>Pattern*</td>
<td>3</td>
<td>4A</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>Mask*</td>
<td>3</td>
<td>4E</td>
<td>4C</td>
<td></td>
</tr>
<tr>
<td>Interrupt Vector*</td>
<td>2</td>
<td>5A</td>
<td>58</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Status/Save registers, Non-chainable</strong></th>
<th>Segment/Tag</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status register</td>
<td>2E</td>
<td>2C</td>
</tr>
<tr>
<td>Interrupt Save register</td>
<td>2A</td>
<td>28</td>
</tr>
</tbody>
</table>

*Slow-readable registers.*
The interface signals and pin assignments are listed in Table 2. Some of the signals are three-state, i.e., they are high-impedance when not under bus control. The open-drain pins require a pullup resistor of 3.3K ohms or more. The DTC decodes the status lines (S70-S73) for the Interrupt Acknowledge signal and generates status for data transactions. The multiplexed input CS/WAIT serves as an active Low Chip Select (CS) signal when the DTC is a bus slave, and serves as an active Low Wait (WAIT) signal when the DTC is bus master and the control bit in the Master Mode register is enabled. The multiplexed output SN7/MMUSYNC is driven Low when the DTC is not in control of the system bus and the MM1 bit of the Master Mode register is set. SN7/MMUSYNC floats to a high-impedance state when the DTC is not in control of the system bus and the MM1 bit is cleared. When the DTC is in control of the system bus and is operating in logical address space, this line outputs an active High MMUSYNC pulse prior to each memory transaction cycle. In physical address space, this line outputs SN7, which is the 24th address bit in the linear address space.

If a peripheral device requires DMA service, it issues a request to the DTC by asserting DREQ. If the channel receiving the request is enabled and the BUSREQ and BAT lines are High, the DTC issues a bus request to the CPU by driving the BUSREQ line Low. When the CPU relinquishes bus control, a Bus Acknowledge signal is output to the DTC by driving the BAI line Low, indicating that the request for bus control has been granted. Upon receipt of the Bus Acknowledge signal, the DTC issues a DMA Acknowledge signal to the peripheral by lowering the DACK output; it then issues the control signals and addresses necessary to effect the transfer. When the transfer is completed or terminated, DACK is driven High and the DTC begins the termination procedure. The DACK output can be programmed as level or pulsed for Flyby transactions and as level or inactive for Flowthrough transactions via the CM18 bit of the Channel Mode register.
<table>
<thead>
<tr>
<th>Interface Signal</th>
<th>Pin Number</th>
<th>Input/Output</th>
<th>Three-State</th>
<th>Open-Drain</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD0-AD15</td>
<td>5-20</td>
<td>In/Out</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>AS</td>
<td>44</td>
<td>In/Out</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>BAI</td>
<td>1</td>
<td>In</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>BAO</td>
<td>3</td>
<td>Out</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>BUSREQ</td>
<td>2</td>
<td>In/Out</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>B/W</td>
<td>35</td>
<td>Out</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CS/WAIT</td>
<td>42</td>
<td>In</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>DACK1, DACK2</td>
<td>39, 40</td>
<td>Out</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>DREQ1, DREQ2</td>
<td>36, 37</td>
<td>In</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>DS</td>
<td>43</td>
<td>In/Out</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>EOP</td>
<td>38</td>
<td>In/Out</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>ICE</td>
<td>46</td>
<td>In</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>IEO</td>
<td>48</td>
<td>Out</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>INT</td>
<td>47</td>
<td>Out</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>N/S</td>
<td>30</td>
<td>Out</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>R/W</td>
<td>41</td>
<td>In/Out</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SN0-SN6</td>
<td>21-25, 28, 29</td>
<td>Out</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SN7/MMUSYNC</td>
<td>27</td>
<td>Out</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>ST0-ST3</td>
<td>31-34</td>
<td>In/Out</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>CLK</td>
<td>45</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GND</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+5V</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To establish DMA operation, the internal registers can be loaded under software by the CPU. The registers are addressed via the low byte of the Address/Data bus (AD7-AD0). The high byte of the Address/Data bus (AD15-AD8) is decoded with the user's chip select logic. Chip Select (CS) must be valid prior to the rising edge of AS to allow the CPU to write to, or read from, the DTC's registers. During a DMA transfer, the DTC generates control signals (R/W, B/W, N/S, and ST0-ST3) to indicate the transfer direction, the data size, and the type of space and transaction. It also generates AS, DS, DACK, and MMUSYNC signals to synchronize timing and to demultiplex the Address/Data lines. Additionally, it generates addresses (SN7-SN0 and AD15-AD0 for physical addressing space or SN6-SN0 and AD15-AD0 for logical addressing space) of the source and destination of the transfer; samples the DREQ, WAIT, and EOP lines; stores the data for the Flow-through transaction; and issues an EOP low signal when the transfer is terminated. Upon termination, the DTC performs either an interrupt, base-to-current reloading, chaining, or does nothing, under the control of Channel Mode register (i.e., bits CM7-CM15).

To relinquish bus control, the DTC drives its BUSREQ line High and allows BAO to follow BAI. The CPU regains bus control upon sampling its BUSREQ input; if inactive, the CPU drives its BUSACK output inactive. Whenever both BAI and BUSREQ are High and no DMA requests are pending, the DTC passes the High signal through BAO to the lower-priority device, enabling it to request bus control. This procedure allows the CPU to regain bus control whenever an interrupting device releases bus control. See the Zilog 1982/83 Data Book* for more details on the Zilog Z-BUS.

**INITIALIZATION**

After a hardware reset (i.e., AS and DS are simultaneously Low) or a software reset (i.e., a reset command is issued to the Command register), take the following steps to initialize the system:

- Clear the Master Mode (MM) register to disable the DTC.
- Set the Chain Abort (CA) and Non-Auto Chaining (NAC) bits in each channel's Status register.
- Load each channel's Chain Address register.
- Issue Start Chain command.

*(document number 00-2034-02)
to minimize interaction with the host CPU, the DTC loads its own control parameters from memory into each channel (i.e., performs chaining). The CPU need to only program the Master Mode register and each channel's Chain Address register (Figure 5). All other registers are loaded by the channels themselves from a reload table located in system memory and pointed to by the Chain Address register. During chaining, the N/S and B/W lines are driven low and the ST3-ST0 outputs are set to 1000 (i.e., Memory Transaction for Data).

The first word in the reload table, the reload word, specifies which registers in the channel are to be reloaded. Bits 0 through 9 in the reload word relate to either one or two registers in the channel (Table 3). When a reload word bit is 1, the register or registers corresponding to that bit are reloaded. The data loaded into the selected registers follow the reload word in memory at successively larger addresses.

The reload table is of variable length. For example, when the contents of the segment and offset fields of Channel 1's Chain Address register are 0000H and 1020H, the reload table is started at location 1020H. Thus, the data stored at location 1020H is the reload word. If the reload word is 03F7H, all of Channel 1's registers are loaded with the data in locations 1022H through 1042H (a total of 17 words). If the reload word is 0203H, only Current Address register A (Current ARA), Channel Mode register, and Chain Address register are reloaded with the data in locations 1022H through 102CH (a total of six words), and the remaining registers are not changed. When loading the address registers, the segment and tag word must precede the offset word (e.g., the segment and tag word of Current Address register A is located at 1022H, while the offset word is located at 1024H).

After the Master Mode bit MM0 is set, a Start Chain command causes the selected channel to clear the NAC bit in its Status register and to start chaining. The control parameters of the channel are reloaded and the channel is ready to perform the DMA operation. DMA operation can be initiated in one of the following three ways:

- By software request--issue a Set Software Request command.
- By hardware request--apply a Low signal on the channel's DREQ input; the Hardware Request Mask bit (CM19) in the Channel Mode register must be cleared.
- By chaining--load a Software Request bit (CM20 = 1) into the Channel Mode register during chaining.

```
0100  7101  0000  LD  R1,#0000  ;RESET
0104  3B16  002C  OUT  %002C,R1
0108  8007  NOP
010A  2101  0000  LD  R1,#0000
010C  3B16  0026  OUT  %0026,R1
0112  8007  NOP
0114  2101  1020  LD  R1,#1020
0118  3B16  0022  OUT  %0022,R1
011C  8007  NOP
011E  2101  0001  LD  R1,#0001
0122  3B16  0038  OUT  %0038,R1
0126  8007  NOP
0128  2101  0030  LD  R1,#0030
012C  3B16  002C  OUT  %002C,R1
0130  8007  NOP
```

Figure 5. Initialization of the Z8016 DTC
When DMA operation is initiated by either software or hardware request, the DTC drives the BUSREQ line Low and performs the DMA operation after it receives an active Low BAI signal. When DMA operation is initiated by chaining, the DTC performs the DMA operation as soon as chaining ends if the MM2 bit (CPU Interleave Enable bit) is clear. If the MM2 bit is set, the channel gives up bus control after chaining and before DMA operation.

**DMA OPERATIONS**

There are three types of DMA operation: transfer, transfer-and-search, and search, each of which can occur in either a Flowthrough or Flyby transaction. They are controlled by programming bits 0 through 3 of the Channel Mode register. The Flip bit (CM4) is used to control the transfer direction. Figure 6 shows state diagrams for the various types of operations. Table 4 lists the operation codes.

Flowthrough Transfer and Flowthrough Transfer-and-Search operations consist of both read and write transactions. When bit CM4 is clear, the DTC reads data from the location specified by The Current Address Register A (ARA) (i.e., the source), stores the data in the Temporary register, compares the data with the unmasked pattern, and then writes the data into the location specified by the Current Address Register B (ARB) (i.e., the destination). When bit CM4 is set, the source location is specified by the

<table>
<thead>
<tr>
<th>Memory</th>
<th>Data</th>
<th>Register</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1020</td>
<td>03FF</td>
<td>Chain Control register</td>
<td>Chaining all registers</td>
</tr>
<tr>
<td>1022</td>
<td>0000</td>
<td>Segment/Tag of Current Address Register A</td>
<td>System data mem, increment, 0 waits</td>
</tr>
<tr>
<td>1024</td>
<td>1F00</td>
<td>Offset of Current Address Register A</td>
<td>Starting address</td>
</tr>
<tr>
<td>1026</td>
<td>0074</td>
<td>Segment/Tag of Current Address Register B</td>
<td>I/O, hold, 2 waits</td>
</tr>
<tr>
<td>1028</td>
<td>FF01</td>
<td>Offset of Current Address Register B</td>
<td>Peripheral address</td>
</tr>
<tr>
<td>102A</td>
<td>00A0</td>
<td>Current Op-Count</td>
<td>160 transfers</td>
</tr>
<tr>
<td>102C</td>
<td>0000</td>
<td>Segment/Tag of Base Address Register A</td>
<td>System data, increment, 0 waits</td>
</tr>
<tr>
<td>102E</td>
<td>2F00</td>
<td>Offset of Base Address Register A</td>
<td>Starting address</td>
</tr>
<tr>
<td>1030</td>
<td>0074</td>
<td>Segment/Tag of Base Address Register B</td>
<td>I/O, hold, 2 waits</td>
</tr>
<tr>
<td>1032</td>
<td>FF01</td>
<td>Offset of Base Address Register B</td>
<td>Peripheral address</td>
</tr>
<tr>
<td>1034</td>
<td>0100</td>
<td>Base Op-Count Register</td>
<td>256 transfers</td>
</tr>
<tr>
<td>1036</td>
<td>1234</td>
<td>Pattern register</td>
<td>0000100100110100 as pattern</td>
</tr>
<tr>
<td>1038</td>
<td>F000</td>
<td>Mask register</td>
<td>1111000000000000 as mask</td>
</tr>
<tr>
<td>103A</td>
<td>0002</td>
<td>Interrupt Vector register</td>
<td>Vector = 02</td>
</tr>
<tr>
<td>103C</td>
<td>0004</td>
<td>Channel Mode High</td>
<td>Pulsed DACK</td>
</tr>
<tr>
<td>103E</td>
<td>0D42</td>
<td>Channel Mode Low</td>
<td></td>
</tr>
<tr>
<td>1040</td>
<td>0000</td>
<td>Segment/Tag of Chain Address</td>
<td></td>
</tr>
<tr>
<td>1042</td>
<td>1080</td>
<td>Offset of Chain Address</td>
<td>Address of next chain control word</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>1080</td>
<td>0182</td>
<td>Chain Control register</td>
<td>Chaining three registers</td>
</tr>
<tr>
<td>1082</td>
<td>0076</td>
<td>Segment/Tag of Current Address Register B</td>
<td>I/O, hold, 4 waits</td>
</tr>
<tr>
<td>1084</td>
<td>FF02</td>
<td>Offset of Current Address Register B</td>
<td>Peripheral address</td>
</tr>
<tr>
<td>1086</td>
<td>0050</td>
<td>Current Op-Count</td>
<td>80 transfers</td>
</tr>
<tr>
<td>1088</td>
<td>0010</td>
<td>Channel Mode High</td>
<td>Software request during chaining</td>
</tr>
<tr>
<td>108A</td>
<td>0240</td>
<td>Channel Mode Low</td>
<td>Interrupt at TC, Address Register A to Address Register B Demand/Bus release, word-to-word flyby</td>
</tr>
</tbody>
</table>
Figure 6a. Flowthrough Transfer and Flowthrough Transfer-and-Search Operations
Current ARB, and the destination is specified by the Current ARA.

Flyby Transfer and Transfer-And-Search operations consist of a single Read cycle or a single Write cycle. When CM4 is clear, the DTC reads the data from the location specified by the Current ARA and the DACK signal strobes the data to the flyby peripheral. In Transfer-And-Search operations, the data is also stored in the Temporary register and compared with the unmasked pattern.
Figure 6c. Search Operation
Table 4. Operation Codes And Programming Suggestions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operation Code</th>
<th>Size</th>
<th>Suggestions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flowthrough</td>
<td>0</td>
<td>W - W</td>
<td>If $CM_4 = 0$ then ARA to ARB; if $CM_4 = 1$ then ARB to ARA</td>
</tr>
<tr>
<td>Transfer</td>
<td>1</td>
<td>B - B</td>
<td>If $CM_{18} = 0$ then level DACK; if $CM_{18} = 1$ then DACK inactive</td>
</tr>
<tr>
<td>Flyby</td>
<td>2</td>
<td>W - W</td>
<td>If $CM_4 = 0$ then ARA to ARB; if $CM_4 = 1$ then ARB to ARA</td>
</tr>
<tr>
<td>Transfer</td>
<td>3</td>
<td>B - B</td>
<td>If $CM_{18} = 0$ then level DACK; if $CM_{18} = 1$ then pulsed DACK</td>
</tr>
<tr>
<td>Flowthrough</td>
<td>4</td>
<td>W - W</td>
<td>$CM_4, CM_{18}$ same as flowthrough transfer</td>
</tr>
<tr>
<td>Transfer &amp;</td>
<td>5</td>
<td>B - B</td>
<td>$CM_17 = 0$ then stop on no match; if $CM_17 = 1$ then stop on match</td>
</tr>
<tr>
<td>Search</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flyby</td>
<td>6</td>
<td>W - W</td>
<td>$CM_4, CM_{18}$ same as flyby transfer</td>
</tr>
<tr>
<td>Transfer &amp;</td>
<td>7</td>
<td>B - B</td>
<td>$CM_17 = 0$ then stop on no match; if $CM_17 = 1$ then stop on match</td>
</tr>
<tr>
<td>Search</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flowthrough</td>
<td>8</td>
<td>B - W</td>
<td>Byte at ARA, word at ARB</td>
</tr>
<tr>
<td>Funneling</td>
<td>9</td>
<td></td>
<td>If $CM_4 = 0$ then byte-to-word; if $CM_4 = 1$ then word-to-byte</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$CM_{18}$ same as transfer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Operation count = number of words</td>
</tr>
<tr>
<td>Flyby</td>
<td>C</td>
<td>B - W</td>
<td></td>
</tr>
<tr>
<td>Funneling</td>
<td>D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Search</td>
<td>E</td>
<td>W - W</td>
<td>If $CM_4 = 0$ then source at ARA; if $CM_4 = 1$ then at ARB</td>
</tr>
<tr>
<td></td>
<td>F</td>
<td>B - B</td>
<td>$CM_17 = 0$ then stop on no match; if $CM_17 = 1$ then stop on match</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operation Code</th>
<th>Suggestions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Operation</td>
<td>0 0</td>
<td>Each Software Rec. command causes one operation; Each DREQ falling edge causes one operation**</td>
</tr>
<tr>
<td>Demand with Bus Hold</td>
<td>0 1</td>
<td>Each Software Req. command causes block operation***; Operating when DREQ Low; Hold bus when DREQ High</td>
</tr>
<tr>
<td>Demand with Bus Release</td>
<td>1 0</td>
<td>Each software Req. command causes block operation***; Operating when DREQ Low; Release bus when High</td>
</tr>
<tr>
<td>Demand Interleave</td>
<td>1 1</td>
<td>Each Software Req. command causes block operation***; Operating when DREQ Low; Release bus to other channel or CPU after each operation</td>
</tr>
</tbody>
</table>

*CM (Channel Mode) register's bit.
**The DREQ falling edge must meet the timing requirement.
***If MM2 (Master Mode) bit is set (CPU interleave is enabled), the DTC releases the bus after each operation when the channel is not in Bus Hold mode.
When Flip bit CM4 is set, the DTC activates DACK to the flyby peripheral, which enables the data onto the A/D bus, writes the data into the location specified by the Current ARB, stores it in the Temporary register, and compares it with the unmasked pattern.

The Search operation consists of a Read cycle only. The DTC reads data from the source location (specified by the Current ARA when CM4 = 0 and by Current ARB when CM4 = 1), stores the data in the Temporary register, and compares it with the unmasked pattern. No data is written into any location or peripheral. Channel Mode register bits CM17-CM16 are the match control field for programming the Stop condition.

Channel Mode bits CM6-CM5 select the channel's response to the request to start a DMA operation. There are four types of response: single operation, demand dedicated with bus hold, demand dedicated with bus release, and demand interleave. These responses are detailed below. Figure 7 shows flow charts for each of these responses. Interleave operations between the CPU and the DTC, and between DTC channels, are shown in Figure 8.

The setting of bits CM6 and CM5 are described as follows:

a) Single operation (CM6 = 0, CM5 = 0). In response to a software request or active DREQ High-to-Low transition, the channel performs a single DMA iteration. The DTC relinquishes bus control after each transaction unless a second High-to-Low DREQ transition meets the timing requirement.

b) Demand Dedicated with Bus Hold (CM6 = 0, CM5 = 1). In response to a software request, the channel acquires bus control, performs a DMA operation until termination occurs (i.e., TC, MC or EOP occurs), and then relinquishes bus control.

In response to an active Low DREQ, the channel acquires bus control, performs DMA operations while DREQ is active Low, retains bus control when DREQ is High but does nothing, resumes DMA operation when DREQ is Low again and only relinquishes bus control when the operation terminates (i.e., TC, MC or EOP occurs). If the DACK signal is programmed as level (CM18 = 0), it will be active Low from the time the channel acquires bus control to when it relinquishes control.

c) Demand Dedicated with Bus Release (CM6 = 1, CM5 = 0). In response to a software request the channel performs DMA iterations until TC, MC, or EOP occurs. In response to a hardware request, the channel performs DMA iterations until DREQ goes inactive. The contents of the Current Address registers and the Current Operation Count register will not be reloaded until TC, MC, or EOP occurs.

d) Demand Interleave (CM6 = 1, CM5 = 1). Demand Interleave varies, depending on the setting of Master Mode register bit MM2. If MM2 is set (CPU interleave is enabled), the DTC relinquishes bus control after each DMA iteration and then re-requests it. This permits the CPU and other devices to gain bus control during DMA operations. If MM2 is clear (CPU interleave is disabled), control can pass from one channel to the other without releasing bus control. If only one channel is programmed in Demand Interleave mode, the other channel will retain control until termination or until DREQ goes inactive, at which time control is returned to the other channel.

Channel Mode register bit CM18 selects the waveform of DACK. The pulsed DACK (CM18 = 1) is used only in Flyby transactions. It is inactive during Non-Flyby transactions when CM18 is set.

Byte-word funneling allows packing and unpacking of byte data to facilitate high-speed transfers between byte-oriented peripherals and word-organized memory. The funneling option can be used only in Flowthrough transactions. For transfers from a byte source to a word destination, two consecutive byte reads are performed to move data from the source location. These bytes are assembled in the Temporary register. The Temporary register data is then written into the destination location as a word. For word-to-byte funneling, word data is read from the source location into the Temporary register. This word is then written to the destination in two consecutive byte writes. The byte address must be programmed in the Current ARA and the word address must be in the Current ARB. Bit CM4 in the Channel Mode register is used to specify the transfer direction. It is set to 0 to specify byte-to-word funneling and to 1 for word-to-byte funneling. To access the high byte of the word first, bit TG3 of the Current ARB must be cleared. Bit TG3 of the Current ARB is set when accessing the low byte of the word first, after which the ARB address increments. Figure 9 shows two examples of data funneling.
Figure 7. Flow Charts of DMA Operations
Figure 8. Flow Charts of Interleave Operations

CH 1: INTERLEAVE
CH 2: SOFTWARE DEMAND
CPU: INTERLEAVE

CH 1: SOFTWARE DEMAND
CH 2: BUS RELEASE
CPU: NO INTERLEAVE

CH 1: SOFTWARE DEMAND
CH 2: BUS HOLD
CPU: INTERLEAVE

CH 1: SOFTWARE DEMAND
CH 2: BUS HOLD OR RELEASE
CPU: INTERLEAVE

Figure 8. Flow Charts of Interleave Operations
A) Byte-to-Word Funneling: Data is moved from the byte source addressed at 0010-FA70 to the word destination addressed from 1600.

Current ARA: 0010-FA70 (Segment = 00, Offset = FA70, Address hold)
Current ARB: 00xx-1604 (Segment = 00, Offset = 1604, Address hold/change)
Current Op-Count: 0003 (Three words)
Flip bit (CM₄): 0 (Data from "ARA" to "ARB")

Source Data String

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>00</th>
<th>01</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>00-1600</td>
<td>*</td>
<td>FFE</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>00-1602</td>
<td>*</td>
<td>DCCC</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>00-1604</td>
<td>AABB</td>
<td>BBAA</td>
<td>EEFF</td>
<td>FFE</td>
</tr>
<tr>
<td>00-1606</td>
<td>CCD</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>00-1608</td>
<td>EEFF</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>00-160A</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

**NOTES**

*WRITE FIRST* | *HIGH* | *LOW* | *HIGH* | *LOW*

B) Word-to-Byte Funneling: Data is moved from the word source addressed from 0000-1AO0 to the byte destination addressed from 1800.

Current ARA: 0000-1AO0 (Segment = 00, Offset = 1AO0, Address increment)
Current ARB: 00xx-1800 (Segment = 00, Offset = 1800, Address hold/change)
Current Op-Count: 003 (three words)
Flip bit (CM₄): 1 (Data from "ARB" to "ARA")

Source Data Distribution

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>Word Data</th>
<th>TG₄, TG₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>00-17FA</td>
<td>6677</td>
<td>00-1A00</td>
</tr>
<tr>
<td>00-17FA</td>
<td>8899</td>
<td>00-1A01</td>
</tr>
<tr>
<td>00-1800</td>
<td>AABB</td>
<td>00-1A02</td>
</tr>
<tr>
<td>00-1802</td>
<td>CCD</td>
<td>00-1A03</td>
</tr>
<tr>
<td>00-1804</td>
<td>EEFF</td>
<td>00-1A04</td>
</tr>
<tr>
<td>00-1806</td>
<td></td>
<td>00-1A05</td>
</tr>
</tbody>
</table>

*Data unchanged

**NOTES**

*READ FIRST* | *HIGH* | *LOW* | *HIGH* | *LOW*

**Figure 9. Examples of Byte/Word Funneling**
Z8016 DTC-TO-Z8000 CPU INTERFACE

CPU and DTC On Same Board

The Address/Data bus and control signals of the Z8000 CPU and those of the Z8016 DTC are directly connected. The AS, DS, and BUSACK signals of the CPU are connected through the reset logic to the AS, DS, and BAI signals of the DTC. CS/WAIT demultiplexing logic is required for the CS/WAIT input of the DTC if hardware waits are necessary. The DREQ lines are connected to the request outputs of peripheral devices. The DACK lines are connected to the corresponding enable inputs of the peripheral devices.

When programming for Flyby transactions, the R/W input of the flyby peripheral should be inverted internally by the peripheral or externally by special logic. R/W High indicates that the flyby peripheral should accept data, and R/W Low indicates that the flyby peripheral should drive data onto the bus. The memory or non-flyby peripheral uses the R/W High signal to indicate that it should drive data onto the A/D bus, and it uses the R/W Low signal to indicate that it should accept the data from A/D bus.

When reading a slow-readable register (e.g., the Channel Mode register), external logic for inserting hardware Wait states is required. The worst-case DS low width for the slow-readable registers is approximately 2000 ns for a 4 MHz Z8016 DTC. The interrupt vector is supplied by the Interrupt Save register (a fast-readable register), therefore, the DS low width for Interrupt Acknowledge does not require hardware Wait states.

CPU and DTC on Different Boards

When the DTC and CPU are located on different boards, the address/data and control signals pass through the system bus. The system bus must provide:

- Multiplexed Address/Data lines (AD0-AD15)
- Bus timing lines [Address Strobe (AS), Data Strobe (DS)]
- Read/Write (R/W) status signal
- Bus control lines [Bus Request (BUSREQ) and Bus Acknowledge (BUSACK)]
- Interrupt Request lines
- Status lines (ST0-ST3)
- Ready (RDY) line

The BUSREQ pin of the DTC requires special bidirectional buffer logic to prevent competition between buses. The other connections are the same as those made when the CPU and DTC are located on the same board.

Figure 10 shows the interface of the Z8000 CPU and the Z8016 DTC when located on the same board. No buffer is required for BUSREQ. The pins of BUSREQ, EOP and INT require 3.3k or larger pullup resistors. When more than one DTC or other peripherals are used, the BAI-BAO and IEl-IEO daisy chains are used to determine priorities for bus control and the interrupt service.

Figure 11 shows the interface configuration for a Z-BUS system used with the Z8016 DTC.
Figure 10. DTC-to-Z8000 CPU Interface Configuration
Figure 11. DTC-to-Z-BUS System Interface Configuration
Z8016 DTC-TO-8086 CPU INTERFACE

To control data transactions the 8086 CPU provides RD and WR signals and the Z8016 DTC provides DS and R/W signals. The R/W signal is valid and stable at the T1 state, whereas RD and WR are valid at the T2 state. Therefore, the use of RD or WR to generate a R/W signal violates the R/W-valid-to-DS falling edge setup time requirement. To avoid this, the DT/R signal of the 8086 CPU can be used to generate the R/W signal for programming the DTC. This interface configuration between the Z8016 DTC and the 8086 CPU is shown in Figure 12.

External logic provides and controls the status signals ST0-ST3. See the Interface Support Logic section of this application note for details.

Z8016 DTC-TO-Z8030 Z-SCC INTERFACE

The Z8030 Serial Communications Controller (Z-SCC) functions as a serial-to-parallel, parallel-to-serial converter/controller. Address and data transactions through the Z-SCC are activated by controlling the CS0 and CS1 inputs. The CS1 must remain active High throughout the data transaction. The CS0 Low allows the address of the internal register to be accessed. Figure 13 shows the DTC-to-Z-SCC interface configuration.

When interfacing with the Z-SCC, the DTC should be programmed for:

- Single operation or Demand operation
- Byte-to-byte flowthrough transfer, transfer-and-search, or search. An FIO is necessary in Flyby mode due to recovery time parameters.
- One wait state insertion for accessing the Z-SCC and three wait states for the memory cycle. This is to meet the SCC recovery time.

For example, to transfer data from the Z-SCC (addressed as 00-FFBx) to memory (e.g., 00-2000 to 00-20FE), the ARA, ARB, Op-Count and Channel Mode registers are:

- ARA: 0000 - 2000
- ARB: 0072 - FFBO
- Op-Count: 0100
- Channel Mode: 0000 - 1001

Because of the write to DS falling edge setup time requirement, Flyby transactions are not recommended unless the memory access time is fast enough to meet this requirement. The Z-SCC requests a DMA transfer by pulling the DTR/REQ output Low.

Z8016 DTC-TO-Z8038 Z-FIO INTERFACE

The Z8038 FIFO I/O Port (Z-FIO) provides an asynchronous, 128-byte FIFO buffer. This buffer is expandable in both width and depth. The data transfer logic of the Z-FIO is especially designed to work with DMA controllers in high-speed transfers. Figure 14 shows the DTC-to-Z-FIO interface configuration. The DACK output of the DTC is connected to the DMASTB input of the Z-FIO. When DACK is active Low, it masks the CS for Flyby DMA operations. The following rules apply when programming the DTC to transfer data between the A/D bus and the Z-FIO.

1. The time between the rising edge of DS and the next falling edge of DS in the DTC must meet the valid access recovery time of the Z-FIO. In Demand Block transfer operations, the delay of two DS signals equals approximately two DMA clock cycles. Therefore, Demand Interleave transfer or Single transfer operations are suggested.

2. The pulsed DACK bit (CM18) of the Channel Mode register must be set.

3. For Flowthrough operations, CS of the Z-FIO must be activated.

4. For word-to-word transfers, two FIOs must be used.
Figure 12. Z8016 DTC-to-8086 CPU Interface Configuration
Figure 13. DTC-to-Z-SCC Interface Configuration
Z8016 DTC-TO-Z8010 MMU INTERFACE

The Z8010 Memory Management Unit (MMU) contains a table of access attributes that are individually programmable for each segment. The attributes provided are read-only, System-mode-only, DMA-only, execute-only, and CPU-only. If the MMU detects a memory access that violates one of the attributes of a segment, the MMU interrupts the CPU or DMA to inhibit an illegal memory access.

Figure 15 shows the DTC-to-MMU interface configuration. The MMUSYNC output of the DTC ORed with the BUSACK signal of the CPU is connected to the DMASYNC input of the MMU. The MMUSYNC pin of the DTC is multiplexed with SN7. If bit MM1 of the Master Mode register is set (Logical Addressing mode), this pin outputs an MMUSYNC active High pulse prior to each DMA cycle when the DTC is in control of the system bus; when the DTC is not in control of the system bus it outputs a Low level. If the MM1 is clear (Physical Addressing mode), this pin outputs the SN7 when the DTC is a bus master and is driven with high-impedance off when the DTC is not in control of the system bus.

The SUP output of the MMU is connected to the EOP pin of the DTC so that DMA operation will be terminated whenever a violation is detected.
Figure 15. DTC-to-MMU Interface Configuration
Figure 16 shows the external logic for multiplexing CS and WAIT (or ROY) signals for the CS/WAIT input of the Z8016 DTC. The slow circuit shown assumes a timeout feature such as on the AMZ8127 clock chip. Figure 17 shows the logic for decoding the status lines to generate the MREQ, IORQ, and M/I0 signals.

(A) WAIT, CS Multiplexing Logic

(B) ROY, CS Multiplexing Logic

Figure 16. Multiplexing Logic for CS/WAIT Input

Figure 17. Status Lines Decoding Logic
INTRODUCTION

Zilog's Z8536 Counter/Timer and Parallel I/O Unit (CIO) and Z8036 (Z-CIO) can provide convenient solutions to many microprocessor-based design problems. Their handshake control, bit manipulation, pattern recognition, and interrupt control capabilities extend the range of applications far beyond that of traditional counter/timer and parallel I/O circuits. This application note gives a generalized procedure for initializing the CIO, as well as an initialization example for one particular application. All comments in this document referring to "the CIO" apply to both the Z8036 and Z8536. References to the Z-CIO refer only to the Z8036.

ACCESSING THE REGISTERS

From the programmer's point of view, the only difference between the Z8036 and the Z8536 is the way the registers are accessed. In the Z8036, they are mapped directly into the CPU's I/O address space, and the Right Justified Address (RJA) bit in the Master Interrupt Control register determines which address bits are used to select them. When RJA = 0, bits AD6-AD1 are decoded, and when RJA = 1, bits AD5-ADO are decoded.

The Z8536 uses only A0 and A1 to select the registers and thus occupies only four bytes of I/O address space. The Data registers for each port are accessed directly using A0 and A1. The Control registers (as well as the Data registers) can be accessed using the following two-step sequence with A0 = A1 = 1: first, write the address of the target register to an internal 6-bit pointer register; then read from or write to the target register. An internal state machine determines whether a given access refers to the pointer or the target register.

SOFTWARE RESET

A software reset is performed by writing a 1 to the Reset bit in the Master Interrupt Control register. This causes all control bits to be reset to 0, all port I/O lines to be at high impedance, the Interrupt pin to be inactive, and the Interrupt Enable Output (IEO) pin to follow the Interrupt Enable Input (IEI) pin. A reset disables all functions except a read or write to the Reset bit; therefore the Reset bit must be cleared before any other control bits can be programmed.

INITIALIZATION

Once the CIO has been reset and, in the Z-CIO, the RJA bit has been programmed, it can easily be initialized for a given application by using the procedures outlined in the flowcharts of Figures 1 through 7. These flowcharts are intended to serve more as a logical guide than as a sequential algorithm. The actual sequence of initialization is unimportant, except that a few basic rules must be observed:

- The ports and counter/timers should be enabled only after their functions have been completely specified.
- When Ports A and B are linked, Port B should be enabled before, or simultaneously with, the enabling of Port A. Also, the Port Link Control (PLC) bit in the Master Configuration Control register should be set before either port is enabled.
- The counter/timers should be triggered only after they have been enabled.

- When Counter/Timers 1 and 2 are linked, the functions of both must be specified and the Counter/Timer Link Control (LC) bits (in the Master Configuration Control register) must be programmed before either counter/timer is enabled.

- The Master Interrupt Enable (MIE) bit in the Master Interrupt Control register should be set only after the functions of the CIO’s interrupt sources have been completely specified.

---

**Figure 1. Port A or B Initialization**

**Table 1. Z8036/Z8536 CIO Register Summary**

<table>
<thead>
<tr>
<th>Internal Address (Binary)</th>
<th>Read/Write</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>A5...A0</td>
<td></td>
<td><strong>Main Control Registers</strong></td>
</tr>
<tr>
<td>000000</td>
<td>R/W</td>
<td>Master Interrupt Control</td>
</tr>
<tr>
<td>000001</td>
<td>R/W</td>
<td>Master Configuration Control</td>
</tr>
<tr>
<td>000010</td>
<td>R/W</td>
<td>Port A Interrupt Vector</td>
</tr>
<tr>
<td>000011</td>
<td>R/W</td>
<td>Port B Interrupt Vector</td>
</tr>
<tr>
<td>000100</td>
<td>R/W</td>
<td>Counter/Timer Interrupt Vector</td>
</tr>
<tr>
<td>000101</td>
<td>R/W</td>
<td>Port C Data Path Polarity</td>
</tr>
<tr>
<td>000110</td>
<td>R/W</td>
<td>Port C Data Direction</td>
</tr>
<tr>
<td>000111</td>
<td>R/W</td>
<td>Port C Special I/O Control</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Most Often Accessed Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>001000 *</td>
</tr>
<tr>
<td>001001 *</td>
</tr>
<tr>
<td>001010 *</td>
</tr>
<tr>
<td>001011 *</td>
</tr>
<tr>
<td>001100 *</td>
</tr>
<tr>
<td>001101 R/W</td>
</tr>
<tr>
<td>001110 R/W</td>
</tr>
<tr>
<td>001111 R/W</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Counter/Timer Related Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000 R</td>
</tr>
<tr>
<td>010001 R</td>
</tr>
<tr>
<td>010010 R</td>
</tr>
</tbody>
</table>

* All bits can be read and some bits can be written.
** Also directly addressable in Z8536 using pins A0 and A1.
Table 1. Z8036/Z8536 C10 Register Summary—Continued

<table>
<thead>
<tr>
<th>Internal Address (Binary)</th>
<th>Read/Write</th>
<th>Register Name</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Counter/Timer Related Registers (continued)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>010011</td>
<td>R</td>
<td>Counter/Timer 2 Current Count (LS Byte)</td>
</tr>
<tr>
<td>010100</td>
<td>R</td>
<td>Counter/Timer 3 Current Count (MS Byte)</td>
</tr>
<tr>
<td>010101</td>
<td>R</td>
<td>Counter/Timer 3 Current Count (LS Byte)</td>
</tr>
<tr>
<td>010110</td>
<td>R/W</td>
<td>Counter/Timer 1 Time Constant (MS Byte)</td>
</tr>
<tr>
<td>010111</td>
<td>R/W</td>
<td>Counter/Timer 1 Time Constant (LS Byte)</td>
</tr>
<tr>
<td>011000</td>
<td>R/W</td>
<td>Counter/Timer 2 Time Constant (MS Byte)</td>
</tr>
<tr>
<td>011001</td>
<td>R/W</td>
<td>Counter/Timer 2 Time Constant (LS Byte)</td>
</tr>
<tr>
<td>011010</td>
<td>R/W</td>
<td>Counter/Timer 3 Time Constant (MS Byte)</td>
</tr>
<tr>
<td>011011</td>
<td>R/W</td>
<td>Counter/Timer 3 Time Constant (LS Byte)</td>
</tr>
<tr>
<td>011100</td>
<td>R/W</td>
<td>Counter/Timer 1 Mode Specification</td>
</tr>
<tr>
<td>011101</td>
<td>R/W</td>
<td>Counter/Timer 2 Mode Specification</td>
</tr>
<tr>
<td>011110</td>
<td>R/W</td>
<td>Counter/Timer 3 Mode Specification</td>
</tr>
<tr>
<td>011111</td>
<td>R</td>
<td>Current Vector</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Port A Specification Registers</td>
</tr>
<tr>
<td>100000</td>
<td>R/W</td>
<td>Port A Mode Specification</td>
</tr>
<tr>
<td>100001</td>
<td>R/W</td>
<td>Port A Handshake Specification</td>
</tr>
<tr>
<td>100010</td>
<td>R/W</td>
<td>Port A Data Path Polarity</td>
</tr>
<tr>
<td>100011</td>
<td>R/W</td>
<td>Port A Data Direction</td>
</tr>
<tr>
<td>100100</td>
<td>R/W</td>
<td>Port A Special I/O Control</td>
</tr>
<tr>
<td>100101</td>
<td>R/W</td>
<td>Port A Pattern Polarity</td>
</tr>
<tr>
<td>100110</td>
<td>R/W</td>
<td>Port A Pattern Transition</td>
</tr>
<tr>
<td>100111</td>
<td>R/W</td>
<td>Port A Pattern Mask</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Port B Specification Registers</td>
</tr>
<tr>
<td>101000</td>
<td>R/W</td>
<td>Port B Mode Specification</td>
</tr>
<tr>
<td>101001</td>
<td>R/W</td>
<td>Port B Handshake Specification</td>
</tr>
<tr>
<td>101010</td>
<td>R/W</td>
<td>Port B Data Path Polarity</td>
</tr>
<tr>
<td>101011</td>
<td>R/W</td>
<td>Port B Data Direction</td>
</tr>
<tr>
<td>101100</td>
<td>R/W</td>
<td>Port B Special I/O Control</td>
</tr>
<tr>
<td>101101</td>
<td>R/W</td>
<td>Port B Pattern Polarity</td>
</tr>
<tr>
<td>101110</td>
<td>R/W</td>
<td>Port B Pattern Transition</td>
</tr>
<tr>
<td>101111</td>
<td>R/W</td>
<td>Port B Pattern Mask</td>
</tr>
</tbody>
</table>
BIT PORT INITIALIZATION

SPECIFY DATA DIRECTION OF EACH BIT IN PORT'S DATA DIRECTION REGISTER

NEED ANY INVERTING DATA PATHS?

YES PROGRAM PORT'S DATA PATH POLARITY REGISTER

NO

NEED ANY OPEN DRAIN OUTPUTS OR IN CATCHER INPUTS?

YES PROGRAM PORT'S SPECIAL I/O CONTROL REGISTER

NO

NEED PATTERN MATCH?

YES PERFORM PATTERN RECOGNITION INITIALIZATION (FIGURE 7)

NO

NEED INTERRUPTS?

NO

WRITE INITIAL DATA TO PORT DATA REGISTER IF NECESSARY

ENABLE PORT IN MASTER CONFIGURATION CONTROL REGISTER

RETURN

YES PERFORM INTERRUPT INITIALIZATION (FIGURE 6)

NEED INTERRUPT ON ERROR FEATURE?

YES SET INTERRUPT ON ERROR BIT IN PORT'S COMMAND AND STATUS REGISTER

NO

Figure 2. Bit Port Initialization
Deskew Timers are used only for output ports.

Figure 3. Handshake Port Initialization
Figure 3. Handshake Port Initialization (continued)

Figure 4. Port C Initialization
For linked operation CT's 1 and 2 must both be initialized before they are enabled.

*For linked operation CT's 1 and 2 must both be initialized before they are enabled.

Figure 5. Counter/Timer Initialization
Figure 6. Interrupt Initialization

Interrupt Initialization

Set the no vector bit in master interrupt control register

Need internally generated vector?

Yes

Need vector includes status feature?

Yes

Set the appropriate vector bit in master interrupt control register

No

Program appropriate interrupt vector register

Set appropriate IE bit in command and status register

Return

Figure 7. Pattern Recognition Initialization

Pattern Recognition Initialization

Specify pattern match mode in port's mode spec register

Port type?

Bit port

Need latch on pattern match feature?

Yes

Set latch on pattern match bit in port's mode spec register

No

Program port's pattern polarity register

Program port's pattern transition register

Program port's pattern mask register

Return

Handshake port

Need interrupt on match only feature?

Yes

Set interrupt on match only bit in port's mode spec register

No
APPLICATION EXAMPLE

Figure 8 shows the Z8036 configured to function as:

- An input handshake port
- A priority interrupt controller
- A squarewave generator
- A watchdog timer
- A general-purpose timer

In addition, there are two bits left over to function as bit-addressable output lines. The following sections discuss the specific initialization procedures used to program each of the functions.

If Port A is to place an interrupt vector on the system bus during Interrupt Acknowledge transactions, then the Port A Interrupt Vector register should be programmed with the appropriate value. The Port A interrupt logic is enabled by writing 1s to bits D7 and D6, and a 0 to bit D5 of the Port A Command and Status register. This encoded command sets the Port A Interrupt Enable (IE) bit.

The programmer should specify the correct data direction for the handshake bits, as well as the initial state of RFD. Writing F4 (hexadecimal) to the Port C Data Direction register programs PC3 (RFD) as an output bit, PC2 (ACKIN) as an input bit, and allows PC1 and PC0 to function as bit-addressable output lines. PC0, PC1, and PC3 can be programmed with their initial values by writing to the Port C Data register. In this example, PC3 (RFD) is initially High, signaling that Port A is ready for data.

Port B as a Priority Interrupt Controller

The priority interrupt controller is implemented using the OR-Priority Encoded Vector (OR-PEV) mode of pattern recognition. When any of the six inputs (PB5-PB7) are High, Port B's Pattern Match Flag and Interrupt Pending (IP) bits are set. If no higher priority interrupt sources (e.g., Port A) are under service, and if Port B's interrupts are enabled, the CIO interrupts the CPU. If no higher priority interrupts are pending at the time of the next Interrupt Acknowledge cycle, then Port B places its interrupt vector on the bus. Encoded within this vector is the value of the highest priority interrupt request at Port B (with PB7 as the highest priority input). The CPU can then automatically branch to the appropriate service routine.

To function as a priority interrupt controller, Port B must be specified as a bit port with OR-PEV pattern match; hence a 06H must be loaded into the Port B Mode Specification register. PB1-PB5 and PB7 must be programmed as input bits by writing 1s to bits D1-D5 of the same register specifies the double-buffered mode and allows the port to interrupt the CPU when both the Buffer register and Input Data register are full. Since the ports reset to Interlocked Handshake, the Port A Handshake Specification register need not be programmed in this example.

Port A as an Input Handshake Port

In Figure 8, Port A is an input port with 2-Wire Interlocked Handshake. (The CIO also supports Strobbed Handshake, Pulsed Handshakes, and IEEE 3-Wire Handshake.) Port C provides the handshake control signals, with PC2 as ACKIN (Acknowledge Input) and PC3 as the RFD (Ready For Data) output.

Port A is specified as an input handshake port by writing a 0 to bit D7 and a 1 to bit D6 of the Port A Mode Specification register. Writing a 1 to bit D5 and a 0 to bit D4 of the same register specifies the double-buffered mode and allows the port to interrupt the CPU when both the Buffer register and Input Data register are full. Since the ports reset to Interlocked Handshake, the Port A Handshake Specification register need not be programmed in this example.

226-009 4-147
PNB6 are masked off; FFH is therefore loaded into the Port B Pattern Polarity register, and BEH is loaded into the Port B Pattern Mask register. Transition pattern specifications should not be used in the OR-PEV pattern match mode, so the Port B Pattern Transition register should not be programmed.

The base interrupt vector should be loaded into the Port B Interrupt Vector register, and the Port B interrupt logic is enabled by writing 1s to bits D7 and D6, and a 0 to bit D5 of the Port B Command and Status register. Also, the Port B Vector Includes Status (VIS) bit should be set so that unique vectors can be generated for each of the interrupt sources (this can be done at the same time the MIE bit is set).

Counter/Timer 1 as a Watchdog Timer

In this example, Counter/Timer 1 acts as a watchdog timer, interrupting the CPU whenever a 10 ms interval elapses without the occurrence of a rising edge on its trigger input (PB6). Each time the timer is triggered (i.e., with each rising edge on PB6), it reloads its time constant and begins counting down toward the terminal count. Since the Counter/Timer 1 Time Constant is programmed to provide a timeout interval of 10 ms, a terminal count condition always indicates that at least 10 ms has elapsed since the last rising edge on PB6.

The programmer must set bits D2 and D4 of the Counter/Timer 1 Mode Specification register. Bit D2 is the Retrigger Enable (REB) bit, and D6 is the External Trigger Enable (ETE) bit. All other bits in this register can remain reset to 0. Since PB6 is the designated external trigger input whenever Counter/Timer 1's ETE bit is set, Port B must be programmed as a bit port and PB6 must be programmed as an input bit.

Since Counter/Timer 1 is in the Timer mode (i.e., it does not have an external count input), it counts the pulses of the internal clock signal (PCLK/2). Assuming a 4 MHz PCLK, the Time Constant should be 20,0000H for a 10 ms timeout interval. This can be achieved by loading 4EH to the most-significant byte of Counter/Timer 1's Time Constant, and 20H to the least-significant byte of Counter/Timer 1's Time Constant.

The base interrupt vector should be loaded into the Counter/Timer Interrupt Vector register, and the Counter/Timer 1 interrupt logic is enabled by writing 1s to bits D7 and D6, and a 0 to bit D5 of the Counter/Timer 1 Command and Status register. Also, the Counter/Timer VIS bit should be set so that Counter/Timers 1 and 2 can generate unique vectors. (This can be done at the same time the MIE bit is set.)

Counter/Timer 2 as a Squarewave Generator

While Counter/Timer 1 uses PB6 as its trigger input, Counter/Timer 2 can use PB0 as its output. The squarewave duty cycle is selected by writing a 1 to bit D1 and a 0 to bit D0 of the Counter/Timer 2 Mode Specification register. Setting bits D7 and D6 of the same register specifies the Continuous mode with an external output. Since PB0 is the designated Counter/Timer 2 output whenever Counter/Timer 2's External Output Enable (EOE) bit is set, Port B must be programmed as a bit port and PB0 must be programmed as an output bit.

In the Squarewave mode, the timeout interval should be equal to half the period of the desired squarewave (see the C10 Technical Manual, section 4.2.5, document number 00-2091-01). A frequency of 100 KHz corresponds to a period of 10 μs and, therefore, a timeout interval of 5 μs. With a 4MHz PCLK, the period of the input clock signal (PCLK/2) is 0.5 μs, and therefore the necessary Time Constant is 10H or 00004H. This value should be loaded into the Counter/Timer 2 Time Constant registers. Since the squarewave generator does not interrupt the CPU, there is no need to enable Counter/Timer 2's interrupt logic.

Counter/Timer 3 as a General-Purpose Timer

For Counter/Timer 3 to interrupt the CPU periodically, the user must specify the Continuous mode by setting bit D7 of the Counter/Timer 3 Mode Specification register. All other bits in this register can remain reset to 0. Loading 4E20H to the Counter/Timer 3 Time Constant registers specifies a 10 ms timeout interval. Writing 1s to bits D7 and D6, and a 0 to bit D5 of the Counter/Timer 3 Command and Status register enables the Counter/Timer 3 interrupt logic.
When all of their functions have been completely specified, the ports and counter/timers can be enabled simultaneously by writing F4H to the Master Configuration Control register. At this point, the counter/timers can be started by setting the Gate Command (GCB) and Trigger Command (TCB) bits in each of their Command and Status registers. Finally, setting the MIE bit, along with the appropriate VIS bits, completes the initialization. Table 2 summarizes the initialization sequence for this application example.
### Table 2. Initialization Sequence for Application Example

<table>
<thead>
<tr>
<th>Step</th>
<th>Register Programmed</th>
<th>Address ( \text{AD}_7-\text{AD}_0 )</th>
<th>Hex Value Loaded</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Master Interrupt Control</td>
<td>X0000000*</td>
<td>01</td>
<td>Reset Z-C10.</td>
</tr>
<tr>
<td>2.</td>
<td>Master Interrupt Control</td>
<td>X0000000X</td>
<td>00</td>
<td>Clear Reset.</td>
</tr>
<tr>
<td>3.</td>
<td>Port A Mode Specification</td>
<td>X1000000X</td>
<td>60</td>
<td>Double-buffered input port, interrupt on two bytes.</td>
</tr>
<tr>
<td>4.</td>
<td>Port A Interrupt Vector</td>
<td>X000010X</td>
<td>VV</td>
<td>Interrupt vector depends on user's system.</td>
</tr>
<tr>
<td>5.</td>
<td>Port A Command and Status</td>
<td>X001000X</td>
<td>C0</td>
<td>Port A Interrupt Enable.</td>
</tr>
<tr>
<td>6.</td>
<td>Port C Data Direction</td>
<td>X000110X</td>
<td>F4</td>
<td>( \text{PC}_2 ) is input, ( \text{PC}_0, \text{PC}_1 ) and ( \text{PC}_3 ) are output.</td>
</tr>
<tr>
<td>7.</td>
<td>Port C Data</td>
<td>X001111X</td>
<td>4B</td>
<td>( \text{RFD} ) is initially High. ( \text{PC}_0 ) and ( \text{PC}_1 ) are initially Low.</td>
</tr>
<tr>
<td>8.</td>
<td>Port B Mode Specification</td>
<td>X101000X</td>
<td>06</td>
<td>Bit port, OR-PEV pattern match.</td>
</tr>
<tr>
<td>9.</td>
<td>Port B Data Direction</td>
<td>X101011X</td>
<td>FE</td>
<td>( \text{PB}_0 ) is output. ( \text{PB}_1-\text{PB}_7 ) are input.</td>
</tr>
<tr>
<td>10.</td>
<td>Port B Pattern Polarity</td>
<td>X101101X</td>
<td>FF</td>
<td>Interrupt inputs are active High.</td>
</tr>
<tr>
<td>11.</td>
<td>Port B Pattern Mask</td>
<td>X101111X</td>
<td>BE</td>
<td>( \text{PB}_0 ) and ( \text{PB}_6 ) are masked off.</td>
</tr>
<tr>
<td>12.</td>
<td>Port B Interrupt Vector</td>
<td>X000011X</td>
<td>VV</td>
<td>Interrupt vector depends on user's system.</td>
</tr>
<tr>
<td>13.</td>
<td>Port B Command and Status</td>
<td>X001001X</td>
<td>C0</td>
<td>Port B Interrupt Enable.</td>
</tr>
<tr>
<td>15.</td>
<td>Counter/Timer 1's Time Constant-MSBs</td>
<td>X010110X</td>
<td>4E</td>
<td>Time Constant = (20,000)(_{10}) for a 10 ms timeout.</td>
</tr>
<tr>
<td>16.</td>
<td>Counter/Timer 1's Time Constant-LSBs</td>
<td>X010111X</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

* If the initial state of the RJA bit is unknown, then the first access to the Master Interrupt Control register must be performed with \( \text{AD}_0 = 0 \).
### Table 2. Initialization Sequence for Application Example--Continued

<table>
<thead>
<tr>
<th>Step</th>
<th>Register Programmed</th>
<th>Address AD&lt;sub&gt;7&lt;/sub&gt;-AD&lt;sub&gt;0&lt;/sub&gt;</th>
<th>Hex Value Loaded</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>17.</td>
<td>Counter/T Timer 1 Interrupt Vector</td>
<td>X000100X</td>
<td>VV</td>
<td>Interrupt vector depends on user's system.</td>
</tr>
<tr>
<td>18.</td>
<td>Counter/T Timer 1 Command and Status</td>
<td>X001010X</td>
<td>CO</td>
<td>Counter/T Timer 1 Interrupt Enable.</td>
</tr>
<tr>
<td>20.</td>
<td>Counter/T Timer 2's Time Constant MSBs</td>
<td>X011000X</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>21.</td>
<td>Counter/T Timer 2's Time Constant LSBs</td>
<td>X011001X</td>
<td>OA</td>
<td>Time Constant = (10)&lt;sub&gt;10&lt;/sub&gt; for 5 μs timeout.</td>
</tr>
<tr>
<td>23.</td>
<td>Counter/T Timer 3 Time Constant MSBs</td>
<td>X011010X</td>
<td>4E</td>
<td>Time Constant = (20,000)&lt;sub&gt;10&lt;/sub&gt; for a 10 ms timeout.</td>
</tr>
<tr>
<td>24.</td>
<td>Counter/T Timer 3's Time Constant LSBs</td>
<td>X011011X</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>25.</td>
<td>Counter/T Timer 3 Command and Status</td>
<td>X001100X</td>
<td>CO</td>
<td>Counter/T Timer 3 Interrupt Enable.</td>
</tr>
<tr>
<td>26.</td>
<td>Master Configuration Control</td>
<td>X000001X</td>
<td>F4</td>
<td>Enable all ports and counter/timers.</td>
</tr>
<tr>
<td>27.</td>
<td>Counter/T Timer 1 Command and Status</td>
<td>X001010X</td>
<td>06</td>
<td>Trigger and Gate commands.</td>
</tr>
<tr>
<td>28.</td>
<td>Counter/T Timer 2 Command and Status</td>
<td>X001011X</td>
<td>06</td>
<td>Trigger and Gate commands.</td>
</tr>
<tr>
<td>29.</td>
<td>Counter/T Timer 3 Command and Status</td>
<td>X001100X</td>
<td>06</td>
<td>Trigger and Gate commands.</td>
</tr>
<tr>
<td>30.</td>
<td>Master Interrupt Control</td>
<td>X000000X</td>
<td>8C</td>
<td>Master Interrupt Enable, Port B Vector Includes Status, Counter/T Timer Vector Includes Status.</td>
</tr>
</tbody>
</table>
This application note describes the use of the Z8030 Serial Communications Controller (Z-SCC) with the Z8000™ CPU to implement a communications controller in a Synchronous Data Link Control (SDLC) mode of operation. In this application, the Z8002 CPU acts as a controller for the Z-SCC. This application note also applies to the non-multiplexed l8530.

One channel of the Z-SCC communicates with the remote station in Half Duplex mode at 9600 bits/second. To test this application, two Z8000 Development Modules are used. Both are loaded with the same software routines for initialization and for transmitting and receiving messages. The main program of one module requests the transmit routine to send a message of the length indicated by the 'COUNT' parameter. The other system receives the incoming data stream, storing the message in its resident memory.

### DATA TRANSFER MODES

The Z-SCC system interface supports the following data transfer modes:

- **Polled Mode.** The CPU periodically polls the Z-SCC status registers to determine if a received character is available, if a character is needed for transmission, and if any errors have been detected.

- **Interrupt Mode.** The Z-SCC interrupts the CPU when certain previously defined conditions are met.

- **Block/DMA Mode.** Using the Wait/Request (W/REQ) signal, the Z-SCC introduces extra wait cycles in order to synchronize the data transfer between a controller or DMA and the Z-SCC.

The example given here uses the block mode of data transfer in its transmit and receive routines.

### SDLC PROTOCOL

Data communications today require a communications 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) communications package. SDLC is a subset of the International Standards Organization (ISO) link control called High-Level Data Link Control (HDLC), which is used for international data communications.

SDLC is a bit-oriented protocol (BOP). It differs from byte-control protocols (BCPs), such as Bisync, in that it uses only 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 the data link control is determined by the position 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 which each octet belongs to a particular field. Each message contains: opening flag, address, control, information, Frame Check Sequence (FCS), and closing flag (figure 1).
Both flag fields contain a unique binary pattern, 01111110, which indicates the beginning or the end of the 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 with a particular device, the other devices ignore the message until the next flag character is detected.

The address field contains one or more octets, which are used to select a particular station on the data link. An address of eight 1s is a global address code that selects all the devices on the data link. When a primary station sends a frame, the address field is used to select one of several secondary stations. 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 that is always present.

The information field contains any actual transferred data. This field may be empty or it may contain an unlimited number of octets. However, because of the limitations of the error-checking algorithm used in the frame-check sequence, however, the maximum recommended block size is approximately 4096 octets.

The frame check sequence field follows the information or control field. The FCS is a 16-bit Cyclic Redundancy Check (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 Z8030 Z-SCC contains the circuitry necessary to generate and check the FCS field.

Zero insertion and 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 extra 0 does not affect the data in any way and is deleted by the receiver, thus restoring the original data pattern.

Zero insertion and deletion insures that the data stream will not contain a flag character or abort sequence. Six 1s preceded and followed by 0s indicate a flag sequence character. Seven to fourteen 1s signify an abort; 15 or more 1s indicate an idle (inactive) line. Under these three conditions, zero insertion and deletion are inhibited. Figure 2 illustrates the various line conditions.

### Figure 1. Fields of the SDLC Transmission Frame

<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</td>
<td>10101011</td>
<td>01111011</td>
<td>01111110</td>
<td></td>
<td>01111110</td>
</tr>
</tbody>
</table>

A. **ZERO INSERTION**

ADDRESS = 10101011
CONTROL = 01111111

B. **ABORT CONDITION**

xxxx101111111111111111
ABORT FLAG

C. **IDLE CONDITION**

xxxx111111111111111111

### Figure 2. Bit Patterns for Various Line Conditions
The SDLC protocol differs from other synchronous protocols with respect to frame timing. In Bisync mode, for example, a host computer might temporarily interrupt transmission by sending sync characters instead of data. This suspended condition continues as long as the receiver does not time out. With SDLC, however, it is invalid to send flags in the middle of a frame to idle the line. Such action causes an error condition and disrupts orderly operation. Thus, the transmitting device must send a complete frame without interruption. If a message cannot be transmitted completely, the primary station sends an abort sequence and restarts the message transmission at a later time.

**SYSTEM INTERFACE**

The Z8002 Development Module consists of a Z8002 CPU, 16k words of dynamic RAM, 2k words of EPROM monitor, a Z80A S10 providing dual serial ports, a Z801 CTC peripheral device providing four counter/timer channels, two Z80A PIO devices providing 32 programmable I/O lines, and wire wrap area for prototyping. The block diagram is depicted in Figure 3. Each of the peripherals in the development module is connected in a prioritized daisy chain configuration. The Z-SCC is included in this configuration by tying its IEP line to the IE0 line of another device, thus making it one step lower in interrupt priority compared to the other device.

**Figure 3. Block Diagram of Z8000 DM**
Two Z8000 Development Modules containing Z-SCCs are connected as shown in Figure 4 and Figure 5. The Transmit Data pin of one is connected to the Receive Data pin of the other and vice versa. The Z8002 is used as a host CPU for loading the modules' memories with software routines.

The Z8002 CPU can address either of the two bytes contained in 16-bit words. The CPU uses an even address (16 bits) to access the most significant byte of a word and an odd address for the least significant byte of a word.

When the Z8002 CPU uses the lower half of the Address/Data bus \((AD_0-AD_7\) the least significant byte) for byte read and write transactions during I/O operations, these transactions are performed between the CPU and I/O ports located at odd I/O addresses. Since the Z-SCC is attached to the CPU on the lower half of the A/D bus, its registers must appear to the CPU at odd I/O addresses. To achieve this, the Z-SCC can be programmed to select its internal registers using lines \(AD_1-AD_5\). This is done either automatically with the Force Hardware Reset command in WR9 or by sending a Select Shift Left Mode command to WROB in channel B of the Z-SCC. For this application, the Z-SCC registers are located at I/O port address 'FExx'. The Chip Select signal \((CSO)\) is derived by decoding I/O address 'FE' hex from lines \(AD_0-AD_{15}\) of the controller.

To select the read/write registers automatically, the Z-SCC decodes lines \(AD_1-AD_5\) in Shift Left mode. The register map for the Z-SCC is depicted in Table 1.

---

**Figure 4. Block Diagram of Two Z8000 CPUs**

---

<table>
<thead>
<tr>
<th>Address (hex)</th>
<th>Write Register</th>
<th>Read Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>FE01</td>
<td>WR0B</td>
<td>RR0B</td>
</tr>
<tr>
<td>FE03</td>
<td>WR1B</td>
<td>RR1B</td>
</tr>
<tr>
<td>FE05</td>
<td>WR2</td>
<td>RR2B</td>
</tr>
<tr>
<td>FE07</td>
<td>WR3B</td>
<td>RR3B</td>
</tr>
<tr>
<td>FE09</td>
<td>WR4B</td>
<td></td>
</tr>
<tr>
<td>FE0B</td>
<td>WR5B</td>
<td></td>
</tr>
<tr>
<td>FE0D</td>
<td>WR6B</td>
<td></td>
</tr>
<tr>
<td>FE0F</td>
<td>WR7B</td>
<td></td>
</tr>
<tr>
<td>FE11</td>
<td>B DATA</td>
<td>B DATA</td>
</tr>
<tr>
<td>FE13</td>
<td>WR9</td>
<td></td>
</tr>
<tr>
<td>FE15</td>
<td>WR10B</td>
<td>RR10B</td>
</tr>
<tr>
<td>FE17</td>
<td>WR11B</td>
<td></td>
</tr>
<tr>
<td>FE19</td>
<td>WR12B</td>
<td>RR12B</td>
</tr>
<tr>
<td>FE1B</td>
<td>WR13B</td>
<td>RR13B</td>
</tr>
<tr>
<td>FE1D</td>
<td>WR14B</td>
<td></td>
</tr>
<tr>
<td>FE1F</td>
<td>WR15B</td>
<td>RR15B</td>
</tr>
<tr>
<td>FE21</td>
<td>WR0A</td>
<td>RR0A</td>
</tr>
<tr>
<td>FE23</td>
<td>WR1A</td>
<td>RR1A</td>
</tr>
<tr>
<td>FE25</td>
<td>WR2</td>
<td>RR2A</td>
</tr>
<tr>
<td>FE27</td>
<td>WR3A</td>
<td>RR3A</td>
</tr>
<tr>
<td>FE29</td>
<td>WR4A</td>
<td></td>
</tr>
<tr>
<td>FE2B</td>
<td>WR5A</td>
<td></td>
</tr>
<tr>
<td>FE2D</td>
<td>WR6A</td>
<td></td>
</tr>
<tr>
<td>FE2F</td>
<td>WR7A</td>
<td></td>
</tr>
<tr>
<td>FE31</td>
<td>A DATA</td>
<td>A DATA</td>
</tr>
<tr>
<td>FE33</td>
<td>WR9</td>
<td></td>
</tr>
<tr>
<td>FE35</td>
<td>WR10A</td>
<td>RR10A</td>
</tr>
<tr>
<td>FE37</td>
<td>WR11A</td>
<td></td>
</tr>
<tr>
<td>FE39</td>
<td>WR12A</td>
<td>RR12A</td>
</tr>
<tr>
<td>FE3B</td>
<td>WR13A</td>
<td>RR13A</td>
</tr>
<tr>
<td>FE3D</td>
<td>WR14A</td>
<td></td>
</tr>
<tr>
<td>FE3F</td>
<td>WR15A</td>
<td>RR15A</td>
</tr>
</tbody>
</table>

**Table 1. Register Map**

---

**INITIALIZATION**

The Z-SCC can be initialized for use in different modes by setting various bits in its write registers. First, a hardware reset must be
Figure 5. Z8002 With SCC
performed by setting bits 7 and 6 of WR9 to one; the rest of the bits are disabled by writing a logic zero.

SDLC protocol is established by selecting a SDLC mode, sync mode enable, and a x1 clock in WR4. A data rate of 9600 baud, NRZ encoding, and a character length of eight bits are among the other options that are selected in this example (Table 2).

Note that WR9 is accessed twice, first to perform a hardware reset and again at the end of the initialization sequence to enable interrupts. The programming sequence depicted in Table 2 establishes the necessary parameters for the receiver and transmitter so that they are ready to perform communication tasks when enabled.

Table 2. Programming Sequence for Initialization

<table>
<thead>
<tr>
<th>Register</th>
<th>Value (hex)</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>WR9</td>
<td>C0</td>
<td>Hardware reset</td>
</tr>
<tr>
<td>WR4</td>
<td>20</td>
<td>x1 clock, SDLC mode, sync mode enable</td>
</tr>
<tr>
<td>WR10</td>
<td>80</td>
<td>NRZ, CRC preset to one</td>
</tr>
<tr>
<td>WR6</td>
<td>AB</td>
<td>Any station address e.g. &quot;AB&quot;</td>
</tr>
<tr>
<td>WR7</td>
<td>7E</td>
<td>SDLC flag (01111110) = &quot;7E&quot;</td>
</tr>
<tr>
<td>WR2</td>
<td>20</td>
<td>Interrupt vector &quot;20&quot;</td>
</tr>
<tr>
<td>WR11</td>
<td>16</td>
<td>Tx clock from BRG output, TRxC pin = BRG out</td>
</tr>
<tr>
<td>WR12</td>
<td>CE</td>
<td>Lower byte of time constant = &quot;CE&quot; for 9600 baud</td>
</tr>
<tr>
<td>WR13</td>
<td>0</td>
<td>Upper byte = 0</td>
</tr>
<tr>
<td>WR14</td>
<td>03</td>
<td>BRG source bit = 1 for PCLK as input, BRG enable</td>
</tr>
<tr>
<td>WR15</td>
<td>00</td>
<td>External Interrupt Disable</td>
</tr>
<tr>
<td>WR5</td>
<td>60</td>
<td>Transmit 8 bits/character SDLC CRC</td>
</tr>
<tr>
<td>WR3</td>
<td>C1</td>
<td>Rx 8 bits/character, Rx enable (Automatic Hunt mode)</td>
</tr>
<tr>
<td>WR1</td>
<td>08</td>
<td>RxInt on 1st char &amp; sp. cond., ext int. disable</td>
</tr>
<tr>
<td>WR9</td>
<td>09</td>
<td>MIE, VIS, status Low</td>
</tr>
</tbody>
</table>

The Z8002 CPU must be operated in System mode to execute privileged I/O instructions. So the Flag and Control Word (FCW) should be loaded with system normal (S/N), and the Vectored Interrupt Enable (VIE) bits set. The Program Status Area Pointer (PSAP) is loaded with the address %4400 using the Load Control Instruction (LDCtl). If the Z8000 Development Module is intended to be used, the PSAP need not be loaded by the programmer because the development module's monitor loads it automatically after the NMI button is pressed.

Since VIS and Status Low are selected in WR9, the vectors listed in Table 3 will be returned during the Interrupt Acknowledge cycle. Of the four interrupts listed, only two, Ch A Receive Character Available and Ch A Special Receive Condition, are used in the example given here.

Table 3. Interrupt Vectors

<table>
<thead>
<tr>
<th>Vector (hex)</th>
<th>Address* (hex)</th>
<th>Interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>2B</td>
<td>446E</td>
<td>Ch A Transmit Buffer Empty</td>
</tr>
<tr>
<td>2A</td>
<td>4472</td>
<td>Ch A External Status Change</td>
</tr>
<tr>
<td>2C</td>
<td>4476</td>
<td>Ch A Receive Char. Available</td>
</tr>
<tr>
<td>2E</td>
<td>447A</td>
<td>Ch A Special Receive Condition</td>
</tr>
</tbody>
</table>

*Assuming that PSAP has been set to 4400 hex, "PS Address" refers to the location in the Program Status Area where the service routine address is stored for that particular interrupt.

TRANSMIT OPERATION

To transmit a block of data, the main program calls up the transmit data routine. With this routine, each message block to be transmitted is stored in memory, beginning with location 'TBUF'. The number of characters contained in each block is determined by the value assigned to the 'COUNT' parameter in the main module.

To prepare for transmission, the routine enables the transmitter and selects the Wait On Transmit function; it then enables the wait function. The Wait On Transmit function indicates to the CPU whether or not the Z-SCC is ready to accept data from the CPU. If the CPU attempts to send data to the Z-SCC when the transmit buffer is full, the Z-SCC asserts its Wait line and keeps it low until the buffer is empty. In response, the CPU extends its I/O cycles until the Wait line goes inactive, indicating that the Z-SCC is ready to receive data.
The CRC generator is reset and the Transmit CRC bit is enabled before the first character is sent, thus including all the characters sent to the Z-SCC in the CRC calculation.

The Z-SCC's transmit underrun/EOM latch must be reset sometime after the first character is transmitted by writing a Reset Tx Underrun/EOM command to WRO. When this latch is reset, the Z-SCC automatically appends the CRC characters to the end of the message in the case of an underrun condition.

Finally, a three-character delay is introduced at the end of the transmission, which allows the Z-SCC sufficient time to transmit the last data byte and two CRC characters before disabling the transmitter.

RECEIVE OPERATION

Once the Z-SCC is initialized, it can be prepared to receive the message. First, the receiver is enabled, placing the Z-SCC in Hunt mode and thus setting the Sync/Hunt bit in status register RRO to 1. In Hunt mode, the receiver searches the incoming data stream for flag characters. Ordinarily, the receiver transfers all the data received between flags to the receive data FIFO. If the receiver is in Hunt mode, however, no data transfer takes place until an opening flag is received. If an abort sequence is received, the receiver automatically re-enters Hunt mode. The Hunt status of the receiver is reported by the Sync/Hunt bit in RRO.

The second byte of an SDLC frame is assumed by the Z-SCC to be the address of the secondary stations for which the frame is intended. The Z-SCC provides several options for handling this address. If the Address Search Mode bit D2 in WR3 is set to zero, the address recognition logic is disabled and all the received data bytes are transferred to the receive data FIFO. In this mode, software must perform any address recognition. If the Address Search Mode bit is set to one, only those frames with addresses that match the address programmed in WR6 or the global address (all 1s) will be transferred to the receive data FIFO. If the Sync Character Load Inhibit bit (D1) in WR3 is set to zero, the address comparison is made across all eight bits of WR6. The comparison can be modified so that only the four most significant bits of WR6 need match the received address. This alteration is made by setting the Sync Character Load Inhibit bit to one. In this mode, the address field is still eight bits wide and is transferred to the FIFO in the same manner as the data. In this application, the address search is performed.

When the address match is accomplished, the receiver leaves the Hunt mode and establishes the Receive Interrupt on First Character mode. Upon detection of the receive interrupt, the CPU generates an Interrupt Acknowledge Cycle. The Z-SCC returns the programmed vector %2C. This vector points to the location %4472 in the Program Status Area which contains the receive interrupt service routine address.

The receive data routine is called from within the receive interrupt service routine. While expecting a block of data, the Wait On Receive function is enabled. Receive read buffer RR8 is read and the characters are stored in memory location RBUF. The Z-SCC in SDLC mode automatically enables the CRC checker for all data between opening and closing flags and ignores the Receive CRC Enable bit (D3) in WR3. The result of the CRC calculation for the entire frame in RR1 becomes valid only when the End Of Frame bit is set in RR1. The processor does not use the CRC bytes, because the last two bits of the CRC are never transferred to the receive data FIFO and are not recoverable.

When the Z-SCC recognizes the closing flag, the contents of the Receive Shift register are transferred to the receive data FIFO, the Residue Code (not applicable in this application) is latched, the CRC error bit is latched in the status FIFO, and the End Of Frame bit is set in the receive status FIFO. When the End Of Frame bit reaches the top of the FIFO, a special receive condition interrupt occurs. The special receive condition register RR1 is read to determine the result of the CRC calculation. If the CRC error bit is zero, the frame received is assumed to be correct; if the bit is 1, an error in the transmission is indicated.

Before leaving the interrupt service routine, the Reset Highest IUS (Interrupt Under Service), Enable Interrupt on Next Receive Character, and Enter Hunt Mode commands are issued to the Z-SCC.
If receive overrun error is made, a special condition interrupt occurs. The Z-SCC presents vector %2E to the CPU, and the service routine located at address %447A is executed. Register RR1 is read to determine which error occurred. Appropriate action to correct the error should be taken by the user at this point. Error Reset and Reset Highest IUS commands are given to the Z-SCC before returning to the main program so that the other lower-priority interrupts can occur.

In addition to searching the data stream for flags, the receiver also scans for seven consecutive 1s, which indicates an abort condition. This condition is reported in the Break/Abort bit (D7) in RRO. This is one of many possible external status conditions. As a result transitions of this bit can be programmed to cause an external status interrupt. The abort condition is terminated when a zero is received, either by itself or as the leading zero of a flag. The receiver leaves Hunt mode only when a flag is found.

SOFTWARE

Software routines are presented in the following pages. These routines can be modified to include various other options (e.g., SDLC Loop, Digital Phase Locked Loop etc.). By modifying the WR10 register, different encoding methods (e.g., NRZI, FMD, FM1) other than NRZ can be used.
## Appendix

### Software Routines

**plasm 1.3**

<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ CODE</th>
<th>STMT</th>
<th>SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>7601</td>
<td>LDA</td>
<td>R1,PSAREA</td>
</tr>
<tr>
<td>0002</td>
<td>4400</td>
<td>LDCTL</td>
<td>PSAPOFF,R1</td>
</tr>
<tr>
<td>0006</td>
<td>2100</td>
<td>LD</td>
<td>R0,%5000</td>
</tr>
<tr>
<td>0008</td>
<td>5000</td>
<td>LD</td>
<td>R1(,%1C),R0</td>
</tr>
<tr>
<td>000C</td>
<td>0010</td>
<td>LD</td>
<td>R0,%76</td>
</tr>
<tr>
<td>0010</td>
<td>7600</td>
<td>LDA</td>
<td>R0,SPCOND</td>
</tr>
<tr>
<td>0012</td>
<td>3310</td>
<td>LD</td>
<td>R1(%76A),R0</td>
</tr>
<tr>
<td>0014</td>
<td>0076</td>
<td>LDA</td>
<td>R0,PSAREA</td>
</tr>
<tr>
<td>0016</td>
<td>7600</td>
<td>LD</td>
<td>R0,SPCOND</td>
</tr>
<tr>
<td>0018</td>
<td>00FA</td>
<td>LD</td>
<td>R1(%76A),R0</td>
</tr>
<tr>
<td>001A</td>
<td>3310</td>
<td>CALL</td>
<td>INIT</td>
</tr>
<tr>
<td>001E</td>
<td>5F00</td>
<td>CALL</td>
<td>TRANSMIT</td>
</tr>
<tr>
<td>0024</td>
<td>00BC</td>
<td>JR</td>
<td>$</td>
</tr>
<tr>
<td>0026</td>
<td>89FF</td>
<td>TBUR</td>
<td>$AB</td>
</tr>
<tr>
<td>0028</td>
<td>AB</td>
<td>BVAL</td>
<td>$AB</td>
</tr>
<tr>
<td>0029</td>
<td>48</td>
<td>BVAL</td>
<td>'E'</td>
</tr>
<tr>
<td>002A</td>
<td>45</td>
<td>BVAL</td>
<td>'E'</td>
</tr>
<tr>
<td>002B</td>
<td>4C</td>
<td>BVAL</td>
<td>'L'</td>
</tr>
<tr>
<td>002C</td>
<td>4C</td>
<td>BVAL</td>
<td>'L'</td>
</tr>
<tr>
<td>002D</td>
<td>4F</td>
<td>BVAL</td>
<td>'O'</td>
</tr>
<tr>
<td>002E</td>
<td>20</td>
<td>BVAL</td>
<td>' '</td>
</tr>
<tr>
<td>002F</td>
<td>54</td>
<td>BVAL</td>
<td>'T'</td>
</tr>
<tr>
<td>0030</td>
<td>48</td>
<td>BVAL</td>
<td>'H'</td>
</tr>
<tr>
<td>0031</td>
<td>45</td>
<td>BVAL</td>
<td>'E'</td>
</tr>
<tr>
<td>0032</td>
<td>52</td>
<td>BVAL</td>
<td>'R'</td>
</tr>
<tr>
<td>0033</td>
<td>45</td>
<td>BVAL</td>
<td>'E'</td>
</tr>
</tbody>
</table>

**plasm 1.3 LOC OBJ CODE $LISTON $TTY $CONSTANT**

| WROA | $FE21 | BASE ADDRESS FOR WRO CHANNEL A1 |
| RROA | $FE21 | BASE ADDRESS FOR RRO CHANNEL A1 |
| RBUF | $4400 | BUFFER AREA FOR RECEIVE CHARACTER |
| PSAREA | $4400 | START ADDRESS FOR PROGRAM STAT AREA |
| COUNT | 12 | NO. OF CHAR. FOR TRANSMIT ROUTINE |

**GLOBAL MAIN PROCEDURE ENTRY**

| 0000 | 7601 | LDA | R1,PSAREA |
| 0002 | 4400 | LDCTL | PSAPOFF,R1 |
| 0006 | 2100 | LD | R0,%5000 |
| 0008 | 5000 | LD | R1(,%1C),R0 |
| 000C | 0010 | LD | R0,%76 |
| 0010 | 7600 | LDA | R0,SPCOND |
| 0012 | 3310 | LD | R1(%76A),R0 |
| 0014 | 0076 | LDA | R0,PSAREA |
| 0016 | 7600 | LD | R0,SPCOND |
| 0018 | 00FA | LD | R1(%76A),R0 |
| 001A | 3310 | CALL | INIT |
| 001E | 5F00 | CALL | TRANSMIT |
| 0024 | 00BC | JR | $ |
| 0026 | 89FF | TBUR | $AB |
| 0028 | AB | BVAL | $AB |
| 0029 | 48 | BVAL | 'E' |
| 002A | 45 | BVAL | 'E' |
| 002B | 4C | BVAL | 'L' |
| 002C | 4C | BVAL | 'L' |
| 002D | 4F | BVAL | 'O' |
| 002E | 20 | BVAL | ' ' |
| 002F | 54 | BVAL | 'T' |
| 0030 | 48 | BVAL | 'H' |
| 0031 | 45 | BVAL | 'E' |
| 0032 | 52 | BVAL | 'R' |
| 0033 | 45 | BVAL | 'E' |

**END MAIN**
********** INITIALIZATION ROUTINE FOR I-SCC **********

```
0038 004E' ENTRY
003A 0047 ALOOP; LD R1, WROA
0042 A920 OUTIB $R1, $R2, R0
0044 3A22 TEST R0
0046 0018 JR R2, ALOOP
0044 2101
0034 2100
0036 000F
0038 7602
003A 004E' ENTRY
003C 2101
003E FE21
0040 0029
0042 A920
0044 3A22
0044 0018
0044 8D04
004A EEF8
004C 9E08
004A 12 SCCTAB

END

********** RECEIVE ROUTINE **********

```

```
**TRANSMIT ROUTINE**

SEND A BLOCK OF EIGHT DATA CHARACTERS
THE BLOCK STARTS AT LOCATION TBUF

008C  GLOBAL TRANSFER ENTRY
008C 2102  LOAD R2,#TBUF  PTR TO START OF BUFFER
008C 0028'  LOAD R0,#68
0090 3A86  OUTB WROA+10,RLO  ENABLE TRANSMITTER
0094 FE28  LOAD R0,#00  WAIT ON TRANSMIT
0096 C980  OUTB WROA+2,RLO
0098 3A86  LOAD R0,#68  WAIT ENABLE
009A FE23  OUTB WROA+2,RLO
009C C988  LOAD R0,#68
009E 3A86  OUTB WROA+2,RLO
00A0 FE23  LOAD R0,#68
00A2 C880  OUTB WROA,RLO
00A6 FE21  LOAD R1,#WROA+16  IWR8A SELECTED
00AA FE31  LOAD R0,#1
00AE 0001
00B0 C869  LOAD R0,#69  ISDLC CRC
00B2 3A86  OUTB WROA+10,RLO  IWR8A=TxCRC ENABLE
00B4 FE28  LOAD R0,#C0  RESET TxCRC GENERATOR
00B6 3A22  OUTB RR0A,RLO  ISEN
00B8 0010  LOAD R0,#COUNT-1  CREATE DELAY BEFORE DISABLING
00BA CRC0
00BC 3A86  LOAD R0,#68  ISEND ADDRESS
00BE FE21  LOAD R0,#00
00C0 2100  LOAD R0,#COUNT-1
00C2 000B
00C4 3A22  OUTB RR01,R2,R0  ISEND MESSAGE
00C6 0010
00C8 2100  LOAD R0,#26  CREATE DELAY BEFORE DISABLING
00CA 039E
00CC 0F81  LOAD R0,DJNZ  ITRANSMITTER SO THAT CRC CAN BE
00CE C800  LOAD R0,#0  ISENT
00DD 3A86  OUTB WROA+10,RLO  DISABLE TRANSMITTER
00DE FE28
00DF 9E08  JP  END TRANSMIT

**RECEIVE INT. SERVICE ROUTINE**

00D6  GLOBAL REC PROCEDURE ENTRY
00D6 93F3  PUSH @R15,R3
00DB 93F2  PUSH @R15,R2
00DA 93F1  PUSH @R15,R1
00DC 93F0  PUSH @R15,R0
00DE 3A94  INR R1,RROA  IREAD STATUS REG RROA
00E0 FE21
00E2 A690  BITB R1,#0  ITEST IF Rx CHAR SET
00E4 E602  JR Z,RESET  IYES CALL RECEIVE ROUTINE
00E6 5F00  CALL RECEIVE
00E8 006C'  RESRET:  LOAD R0,#38
00EC C838  OUTB WROA,RLO  IRESET HIGHEST IUS
00FA 3A86  OUTB WROA,RLO
00FE FE21
00FO 97F0  POP R0,#R15
00F2 97F1  POP R1,#R15
00F4 97F2  POP R2,#R15
00F6 97F3  POP R3,#R15
00F8 7800  IRET
00FA  END REC
I ********** SPECIAL CONDITION INTERRUPT SERVICE ROUTINE **********!

GLOBAL SPCOND PROCEDURE
ENTRY

00FA 93F0
    PUSH @R15,R0
00FC 3A84
    INB RL0,RR0A+2
00FE 8E23
    BITB RL0,#7
0100 A687
    IREAD ERRORS!
0102 FB03
    IREAD ERRORS!
0104 C820
    IREAD ERRORS!
0106 3A86
    IREAD ERRORS!
0108 FE21
    IREAD ERRORS!
010A CB10
    IREAD ERRORS!
010C 3A86
    IREAD ERRORS!
010E FE21
    IREAD ERRORS!
0110 CB08
    IREAD ERRORS!
0112 3A86
    IREAD ERRORS!
0114 FE23
    IREAD ERRORS!
0116 CB38
    IREAD ERRORS!
0118 3A86
    IREAD ERRORS!
011A FE21
    IREAD ERRORS!
011C 97F0
    POP R0,@R15
011E 7B00
    IRET

END SPCOND

END SDLC
Zilog

Application Note

October 1982

Zilog's Z8030 Z-SCC Serial Communications Controller is one of a family of components that are Z-BUS™ compatible with the Z8000™ CPU. Combined with a Z8000 CPU (or other existing 8- or 16-bit CPUs with nonmultiplexed buses when using the Z8530 SCC), the Z-SCC forms an integrated data communications controller that is more cost effective and more compact than systems incorporating UARTs, baud rate generators, and phase-locked loops as separate entities.

The approach examined here implements a communications controller in a Binary Synchronous mode of operation, with a Z8002 CPU acting as controller for the Z-SCC.

One channel of the Z-SCC is used to communicate with the remote station in Half Duplex mode at 9600 bits/second. To test this application, two Z8000 Development Modules are used. Both are loaded with the same software routines for initialization and for transmitting and receiving messages. The main program of one module requests the transmit routine to send a message of the length indicated in the 'COUNT' parameter. The other system receives the incoming data stream, storing the message in its resident memory.

DATA TRANSFER MODES

The Z-SCC system interface supports the following data transfer modes:

- Polled Mode. The CPU periodically polls the Z-SCC status registers to determine the availability of a received character, if a character is needed for transmission, and if any errors have been detected.

- Interrupt Mode. The Z-SCC interrupts the CPU when certain previously defined conditions are met.

- Block/DMA Mode. Using the Wait/Request (W/REQ) signal, the Z-SCC introduces extra wait cycles to synchronize data transfer between a CPU or DMA controller and the Z-SCC.

The example given here uses the block mode of data transfer in its transmit and receive routines.

SYNCHRONOUS MODES

Three variations of character-oriented synchronous communications are supported by the Z-SCC: Monosync, Bisync, and External Sync (Figure 1). In Monosync mode, a single sync character is transmitted, which is then compared to an identical sync character in the receiver. When the receiver recognizes this sync character, synchronization is complete; the receiver then transfers subsequent characters into the receiver FIFO in the Z-SCC.

Bisync mode uses a 16-bit or 12-bit sync character in the same way to obtain synchronization. External Sync mode uses an external signal to mark the beginning of the data field; i.e., an external input pin (SYNC) indicates the start of the information field.

Figure 1. Synchronous Modes of Communication
In all synchronous modes, two Cyclic Redundancy Check (CRC) bytes can be concatenated to the message to detect data transmission errors. The CRC bytes inserted in the transmitted message are compared to the CRC bytes computed to the receiver. Any differences found are held in the receive error FIFO.

**SYSTEM INTERFACE**

The Z8002 Development Module consists of a Z8002 CPU, 16K words of dynamic RAM, 2K words of EPROM.

Two Z8000 Development Modules containing Z-SCCs are connected as shown in Figure 3 and Figure 4. The Transmit Data pin of one is connected to the Receive Data pin of the other and vice versa. The Z8002 is used as a host CPU for loading the modules' memories with software routines.

The Z8000 CPU can address either of the two bytes contained in 16-bit words. The CPU uses an even address (16 bits) to access the most-significant byte of a word and an odd address for the least-significant byte of a word.

---

**Figure 2. Block Diagram of Z8000 DM**

*Monitor, a Z80A SIO providing dual serial ports, a Z80A CTC peripheral device providing four counter/timer channels, two Z80A PIO devices providing 32 programmable I/O lines, and wire wrap area for prototyping. The block diagram is depicted in Figure 2. Each of the peripherals in the development module is connected in a prioritized daisy-chain configuration. The Z-SCC is included in this configuration by tying its E1 line to the IEO line of another device, thus making it one step lower in interrupt priority compared to the other device.*

**Figure 3. Block Diagram of Two Z8000 Development Modules**
Figure 4. Z8002 with SCC
When the Z8002 CPU uses the lower half of the Address/Data bus (AD0-AD7 the least significant byte) for byte read and write transactions during I/O operations, these transactions are performed between the CPU and I/O ports located at odd I/O addresses. Since the Z-SCC is attached to the CPU on the lower half of the A/D bus, its registers must appear to the CPU at odd I/O addresses. To achieve this, the Z-SCC can be programmed to select its internal registers using lines AD1-AD5. This is done either automatically with the Force Hardware Reset command in WR9 or by sending a Select Shift Left Mode command to WROB in channel B of the Z-SCC. For this application, the Z-SCC registers are located at I/O port address 'FExx'. The Chip Select signal (CSO) is derived by decoding I/O address 'FE' hex from lines AD8-AD15 of the controller. The Read/Write registers are automatically selected by the Z-SCC when internally decoding lines AD1-AD5 in Shift Left mode. To select the Read/Write registers automatically, the Z-SCC decodes lines AD1-AD5 in Shift Left mode. The register map for the Z-SCC is depicted in Table 1.

**INITIALIZATION**

The Z-SCC can be initialized for use in different modes by setting various bits in its Write registers. First, a hardware reset must be performed by setting bits 7 and 6 of WR9 to one; the rest of the bits are disabled by writing a logic zero.

Asynic mode is established by selecting a 16-bit sync character, Sync Mode Enable, and a X1 clock in WR4. A data rate of 9600 baud, NRZ encoding, and a data character length of eight bits are among the other options that are selected in this example (Table 2).

Note that WR9 is accessed twice, first to perform a hardware reset and again at the end of the initialization sequence to enable the interrupts. The programming sequence depicted in Table 2 establishes the necessary parameters for the receiver and the transmitter so that, when enabled, they are ready to perform communication tasks. To avoid internal race and false interrupt conditions, it is important to initialize the registers in the sequence depicted in this application note.

<table>
<thead>
<tr>
<th>Address (hex)</th>
<th>Write Register</th>
<th>Read Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>FE01</td>
<td>WR0B</td>
<td>RR0B</td>
</tr>
<tr>
<td>FE03</td>
<td>WR1B</td>
<td>RR1B</td>
</tr>
<tr>
<td>FE05</td>
<td>WR2</td>
<td>RR2B</td>
</tr>
<tr>
<td>FE07</td>
<td>WR3B</td>
<td>RR3B</td>
</tr>
<tr>
<td>FE09</td>
<td>WR4B</td>
<td></td>
</tr>
<tr>
<td>FE0B</td>
<td>WR5B</td>
<td></td>
</tr>
<tr>
<td>FE0D</td>
<td>WR6B</td>
<td></td>
</tr>
<tr>
<td>FE0F</td>
<td>WR7B</td>
<td></td>
</tr>
<tr>
<td>FE11</td>
<td>B DATA</td>
<td>B DATA</td>
</tr>
<tr>
<td>FE13</td>
<td>WR9</td>
<td></td>
</tr>
<tr>
<td>FE15</td>
<td>WR10B</td>
<td>RR10B</td>
</tr>
<tr>
<td>FE17</td>
<td>WR11B</td>
<td></td>
</tr>
<tr>
<td>FE19</td>
<td>WR12B</td>
<td>RR12B</td>
</tr>
<tr>
<td>FE1B</td>
<td>WR13B</td>
<td>RR13B</td>
</tr>
<tr>
<td>FE1D</td>
<td>WR14B</td>
<td></td>
</tr>
<tr>
<td>FE1F</td>
<td>WR15B</td>
<td>RR15B</td>
</tr>
<tr>
<td>FE21</td>
<td>WR0A</td>
<td>RR0A</td>
</tr>
<tr>
<td>FE23</td>
<td>WR1A</td>
<td>RR1A</td>
</tr>
<tr>
<td>FE25</td>
<td>WR2</td>
<td>RR2A</td>
</tr>
<tr>
<td>FE27</td>
<td>WR3A</td>
<td>RR3A</td>
</tr>
<tr>
<td>FE29</td>
<td>WR4A</td>
<td></td>
</tr>
<tr>
<td>FE2B</td>
<td>WR5A</td>
<td></td>
</tr>
<tr>
<td>FE2D</td>
<td>WR6A</td>
<td></td>
</tr>
<tr>
<td>FE2F</td>
<td>WR7A</td>
<td></td>
</tr>
<tr>
<td>FE31</td>
<td>A DATA</td>
<td>A DATA</td>
</tr>
<tr>
<td>FE33</td>
<td>WR9</td>
<td></td>
</tr>
<tr>
<td>FE35</td>
<td>WR10A</td>
<td>RR10A</td>
</tr>
<tr>
<td>FE37</td>
<td>WR11A</td>
<td></td>
</tr>
<tr>
<td>FE39</td>
<td>WR12A</td>
<td>RR12A</td>
</tr>
<tr>
<td>FE3B</td>
<td>WR13A</td>
<td>RR13A</td>
</tr>
<tr>
<td>FE3D</td>
<td>WR14A</td>
<td></td>
</tr>
<tr>
<td>FE3F</td>
<td>WR15A</td>
<td>RR15A</td>
</tr>
</tbody>
</table>

The Z8002 CPU must be operated in System mode in order to execute privileged I/O instructions, so the Flag Control Word (FCW) should be loaded with System/Normal (S/N), and the Vectored Interrupt Enable (VIE) bits set. The Program Status Area Pointer (PSAP) is loaded with address %4400 using the Load Control instruction (LDCTL). If the Z8000 Development Module is intended to be used, the PSAP need not be loaded by the programmer as the development modules monitor loads it automatically after the NMI button is pressed.
Table 2. Programming Sequence for Initialization

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>WR9</td>
<td>CO</td>
<td>Hardware reset</td>
</tr>
<tr>
<td>WR4</td>
<td>10</td>
<td>x1 clock, 16-bit sync, sync mode enable</td>
</tr>
<tr>
<td>WR10</td>
<td>0</td>
<td>NRZ, CRC preset to zero</td>
</tr>
<tr>
<td>WR6</td>
<td>AB</td>
<td>Any sync character &quot;AB&quot;</td>
</tr>
<tr>
<td>WR7</td>
<td>CD</td>
<td>Any sync character &quot;CD&quot;</td>
</tr>
<tr>
<td>WR2</td>
<td>20</td>
<td>Interrupt vector &quot;20&quot;</td>
</tr>
<tr>
<td>WR11</td>
<td>16</td>
<td>Tx clock from BRG output, TRxC pin = BRG out</td>
</tr>
<tr>
<td>WR12</td>
<td>CE</td>
<td>Lower byte of time constant = &quot;CE&quot; for 9600 baud</td>
</tr>
<tr>
<td>WR13</td>
<td>0</td>
<td>Upper byte = 0</td>
</tr>
<tr>
<td>WR14</td>
<td>03</td>
<td>BRG source bit = 1 for PCLK as input, BRG enable</td>
</tr>
<tr>
<td>WR15</td>
<td>00</td>
<td>External interrupt disable</td>
</tr>
<tr>
<td>WR5</td>
<td>64</td>
<td>Tx 8 bits/character, CRC=16</td>
</tr>
<tr>
<td>WR3</td>
<td>C1</td>
<td>Rx 8 bits/character, Rx enable (Automatic Hunt mode)</td>
</tr>
<tr>
<td>WR1</td>
<td>08</td>
<td>RxInt on 1st char &amp; sp. cond., ext. int. disable</td>
</tr>
<tr>
<td>WR9</td>
<td>09</td>
<td>MIE, VIS, Status Low</td>
</tr>
</tbody>
</table>

Since VIS and Status Low are selected in WR9, the vectors listed in Table 3 will be returned during the Interrupt Acknowledge cycle. Of the four interrupts listed, only two, Ch A Receive Character Available and Ch A Special Receive Condition, are used in the example given here.

Table 3. Interrupt Vectors

<table>
<thead>
<tr>
<th>Vector</th>
<th>Address</th>
<th>Interrupt</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS</td>
<td>(hex)</td>
<td>(hex)</td>
</tr>
<tr>
<td>2B</td>
<td>446E</td>
<td>Ch A Transmit Buffer Empty</td>
</tr>
<tr>
<td>2A</td>
<td>4472</td>
<td>Ch A External Status Change</td>
</tr>
<tr>
<td>2C</td>
<td>4476</td>
<td>Ch A Receive Char. Available</td>
</tr>
<tr>
<td>2E</td>
<td>447A</td>
<td>Ch A Special Receive Condition</td>
</tr>
</tbody>
</table>

* "PS Address" refers to the location in the Program Status Area where the service routine address is stored for that particular interrupt, assuming that PSAP has been set to 4400 hex.

TRANSMIT OPERATION

To transmit a block of data, the main program calls up the transmit data routine. With this routine, each message block to be transmitted is stored in memory, beginning with location 'TBUF'. The number of characters contained in each block is determined by the value assigned to the 'COUNT' parameter in the main module.

To prepare for transmission, the routine enables the transmitter and selects the Wait On Transmit function; it then enables the wait function. The Wait On Transmit function indicates to the CPU whether or not the Z-SCC is ready to accept data from the CPU. If the CPU attempts to send data to the Z-SCC when the transmit buffer is full, the Z-SCC asserts its Wait line and keeps it low until the buffer is empty. In response, the CPU extends its I/O cycles until the Wait line goes inactive, indicating that the Z-SCC is ready to receive data.

The CRC generator is reset and the Transmit CRC bit is enabled before the first character is sent, thus including all the characters sent to the Z-SCC in the CRC calculation, until the Transmit CRC bit is disabled. CRC generation can be disabled for a particular character by resetting the TxCRC bit within the transmit routine. In this application, however, the Transmit CRC bit is not disabled, so that all characters sent to the Z-SCC are included in the CRC calculation.

The Z-SCC's transmit underrun/EOM latch must be reset sometime after the first character is transmitted by writing a Reset Tx Underrun/EOM command to WR0. When this latch is reset, the Z-SCC automatically appends the CRC characters to the end of the message in the case of an underrun condition.

Finally, a five-character delay is introduced at the end of the transmission, which allows the Z-SCC sufficient time to transmit the last data byte, two CRC characters, and two sync characters before disabling the transmitter.

RECEIVE OPERATION

Once the Z-SCC is initialized, it can be prepared to receive data. First, the receiver is enabled, placing the Z-SCC in Hunt mode and thus
setting the Sync/Hunt bit in status register RRO to 1. In Hunt mode, the receiver is idle except that it searches the incoming data stream for a sync character match. When a match is discovered between the incoming data stream and the sync characters stored in WR6 and WR7, the receiver exits the Hunt mode, resetting the Sync/Hunt bit in status register RRO and establishing the Receive Interrupt On First Character mode. Upon detection of the receive interrupt, the CPU generates an Interrupt Acknowledge cycle. The Z-SCC sends to the CPU vector %2E, which points to the location in the Program Status Area from which the receive interrupt service routine is accessed.

The receive data routine is called from within the receive interrupt service routine. While expecting a block of data, the Wait On Receive function is enabled. Receive data buffer RR8 is read, and the characters are stored in memory locations starting at RBUF. The Start of Text (%02) character is discarded. After the End of Transmission character (%04) is received, the two CRC bytes are read. The result of the CRC check becomes valid two characters later, at which time, RR1 is read and the CRC error bit is checked. If the bit is zero, the message received can be assumed correct; if the bit is 1, an error in the transmission is indicated.

Before leaving the interrupt service routine, Reset Highest IUS (Interrupt Under Service), Enable Interrupt on Next Receive Character, and Enter Hunt Mode commands are issued to the Z-SCC.

If a receive overrun error is made, a special condition interrupt occurs. The Z-SCC presents the vector %2E to the CPU, and the service routine located at address %447A is executed. The Special Receive Condition register RR1 is read to determine which error occurred. Appropriate action to correct the error should be taken by the user at this point. Error Reset and Reset Highest IUS commands are given to the Z-SCC before returning to the main program so that the other lower.Priority interrupts can occur.

SOFTWARE

Software routines are presented in the following pages. These routines can be modified to include various versions of Bisync protocol, such as Transparent and Nontransparent modes. Encoding methods other than NRZ (e.g., NRZI, FMO, FM1) can also be used by modifying WR10.
Software Routines

plzasm 1.3

LOC OBJ CODE STMT SOURCE STATEMENT

$LISTON

$TTY

CONSTANT

WROA := $FE21
RROA := $FE21
BUFF := $5400
PSAREA := $4400
COUNT := 12

GLOBAL MAIN PROCEDURE

ENTRY

LDA R1,PSAREA
LDCTL PSAPOF,R1
LD $5000
LD R1(R1C),RO
LOA RO,REC
LD R1($76),RO
LOA RO,SPCOND
LD R1($7A),RO
CALL INIT
JR $TAU, BVAL '02
BVAL '1'
BVAL '2'
BVAL '3'
BVAL '4'
BVAL '5'
BVAL '6'
BVAL '7'
BVAL '8'
BVAL '9'
BVAL '0'
BVAL '1'

IBASE ADDRESS FOR WRO CHANNEL A1
IBASE ADDRESS FOR RRO CHANNEL A1
BUFFER AREA FOR RECEIVE CHARACTER
ISTART ADDRESS FOR PROGRAM STAT AREA
INO. OF CHAR. FOR TRANSMIT ROUTINE

IBASE ADDRESS FOR WRO CHANNEL AI
IBASE ADDRESS FOR RRO CHANNEL AI
IBUFFER AREA FOR RECEIVE CHARACTER I
ISTART ADDRESS FOR PROGRAM STAT AREA I
INO. OF CHAN. FOR TRANSMIT ROUTINE I

IBVAL MEANS BYTE VALUE. MESSAGE CHAR.
INITIALIZATION ROUTINE FOR Z-SCC

GLOBAL INIT
PROCEDURE ENTRY
LOOP: LO
A00
INC OUTIB TEST JR RET SCCTAB.

END INIT

RECEIVE ROUTINE

GLOBAL RECEIVE PROCEDURE
ENTRY

PROCESS CRC ERROR IF ANY, AND GIVE ERROR RESET COMMAND IN WROA
LD RLO,#0
OUTB WROA+6,RLO

END RECEIVE
*************** TRANSMIT ROUTINE ***************
SEND A BLOCK OF DATA CHARACTERS
THE BLOCK STARTS AT LOCATION TBUF

00A6 GLOBAL TRANSMIT PROCEDURE
00A6 2102 LD R2,#TBUF 'IPTR TO START OF BUFFER!
00A8 0028' LDB RLO,,%6C
00AC C86C OUTB WROA+10,RLO #ENABLE TRANSMITTER!
00AE FE28 OOB RLO,,%00 'WAIT ON TRANSMIT!
00B0 C800 OUTB WROA+2,RLO
00B2 3A86 LDB RLO,,%88
00B4 PE23 OUTB WROA+2,RLO #WAIT ENABLE, INT ON 1ST & SP COND!
00B6 3A86 LDB RLO,,%80
00B8 3A86 OUTB WROA,RLO #RESET TxCRC GENERATOR!
00BA FE23 LD R1,#WROA+16 #WR8A SELECTED!
00BCCB C86D LDB RLO,,%6D
00C0 3A86 OUTB WROA+10,RLO #TxC CRC ENABLE!
00CA FE23 LDB RLO,,%8D
00CE 2100 LD R0,1
00D0 3A22 OTIRB @R1,@R2,R0 #SEND START OF TEXT!
00D2 0010 LDB RLO,,%C0
00D4 C800 OUTB WROA,RLO #RESET TxCUND/EOM LATCH!
00D6 3A86 LDB RLO,,%8A
00D8 FE21 OUTB WROA+2,RLO #ENTRY
00DA 2100 LD R0,#COUNT-1
00DC 000B OUTB WROA+6,RLO #SEND MESSAGE!
00DE 3A22 OIRB @R1,@R2,R0 #SEND MESSAGE!
00E0 0010 LDB RLO,,%04
00E2 C804 OUTB @R1,RLO #SEND END OF TRANSMISSION CHARACTER!
00E4 3E18 LD R0,#1670 #CREATE DELAY BEFORE DISABLING!
00E6 2100 OUTB WROA,RLO #DISABLE TRANSMITTER!
00E8 0686 LDB RLO,0
00EA 00E8 OUTB WROA+10,RLO #DISABLE TRANSMITTER!
00F0 FE2B RET
00F2 9E08 END TRANSMIT

*************** RECEIVE INT. SERVICE ROUTINE ***************

00F4 GLOBAL REC PROCEDURE ENTRY
00F4 93F0 PUSH @R15,R0 #READ STATUS FROM RROA!
00F6 3A84 INB RLO,RROA #TEST IF SYNCH HUNT RESET!
00F8 FE21 BITB RLO,#4 #YES CALL RECEIVE ROUTINE!
00FA A684 CALL RECEIVE #YES CALL RECEIVE ROUTINE!
00FC E802 CALL RECEIVE #YES CALL RECEIVE ROUTINE!
00FE 5F00 CALL RECEIVE #YES CALL RECEIVE ROUTINE!
0100 066C #RESET: LDB RLO,,%08 'WAIT DISABLE!
0102 C808 OUTB WROA+2,RLO #WAIT DISABLE!
0104 3A86 LDB RLO,,%B1
0106 PE23 OUTB WROA+6,RLO #ENTER HUNT MODE!
0108 C8D1 LDB RLO,,%20
010A 3A86 OUTB WROA,RLO #ENABLE INT ON NEXT CHAR!
010C FE27 OUTB WROA,RLO #ENABLE INT ON NEXT CHAR!
0110 3A66 LDB RLO,,%38
0112 FE21 OUTB WROA,RLO #RESET HIGHEST IUS!
0114 C838 LDB RLO,,%38
0116 3A66 OUTB WROA,RLO #RESET HIGHEST IUS!
0118 FE21 LDB RLO,,%38
011A 3F70 POP R0,#R15
011C 7B00 RET
011E END REC

4-173
GLOBAL SPCOND PROCEDURE
ENTRY
011E 93F0
0120 3A84
0122 3E23
0124 CB30
0126 3A86
0128 FE21
012A CB08
012C 3A86
012E FE23
0130 CB01
0132 3A86
0134 FE27
0136 CB38
0138 3A86
013A FE21
013C 97F0
013E 7B00
0140 END SPCOND
END BISYNC

0 errors
Assembly complete
This application note describes the software initialization procedure for the Zilog Serial Communications Controller; the procedure applies to both the Z-SCC (Z8030) and the SCC (Z8530). Although the Z8030 and Z8530 have different bus interfaces, their registers are programmed in the same order.

A worksheet is provided in this application note to assist with the initialization process. A program example of how the Z8000 initializes the SCC for asynchronous operation is shown in Appendix A. Other operation modes are initialized in a similar manner and are described in the SCC Technical Manual (document number 00-2057-01).

Each of the SCC's two channels has its own separate Write registers that are programmed to initialize the different operating modes. There are two types of bits in the Write registers: Mode bits and Command bits. Write Register 14, shown in Figure 1, is an example of a register that contains both types of bits.

Bits D4-D0 are Mode bits that can be enabled or disabled by being set to 1 or reset to 0. Each bit has one function. For example, bit D0 enables and disables the BR generator.

Figure 1. Command and Mode Bits
Bits D7-D5 are Command bits, which require the decoding of several bits to enable the function. (Command bits are usually denoted by having boxes drawn around them—see Figure 1.) Functions controlled by the Command bits can only be enabled; they cannot be toggled like the Mode bits. For example, the Search mode is entered by setting bits D7-D5 to 001. Each command requires a separate write of the entire register. Care must be taken when issuing a command, so that the Mode bits are not changed accidentally.

**INITIALIZATION PROCEDURE**

The SCC initialization procedure is divided into three stages. The first stage consists of programming the operation modes (e.g., bits per character, parity) and loading the constants (e.g., interrupt vector, time constants). The second stage entails enabling the hardware functions (e.g., transmitter, receiver, baud rate generator). It is important that the operating modes are programmed before the hardware functions are enabled. The third stage, if required, consists of enabling the different interrupts.

Table 1 shows the order (from top to bottom) in which the SCC registers are to be programmed. Those registers that need not be programmed are listed as optional in the comments column. The bits in the registers that are marked with an "X" are to be programmed by the user. The bits marked with an "S" are to be set to their previously programmed value. For example, in stage 2, Write Register 3 bits D7-D5 are shown with an "S" because they have been programmed in stage 1 and must remain set to the same value.

**INITIALIZATION TABLE**

Figure 2 provides a worksheet that can be used as an aid when initializing the SCC. The bits that must be programmed as either a 0 or a 1 are filled in; the remaining bits are left blank to be programmed by the user according to the desired mode of operation. The binary value can then be converted to a hexadecimal number and placed in the table after the Write register notation in the column labeled "HEX." When completed, the worksheet in Figure 2 can be used to produce a program initialization table.

**RESET CONDITIONS**

The SCC should be reset by either hardware or software before initialization. A hardware reset can be accomplished by simultaneously grounding RD and WR on the Z8530 or AS and DS on the Z8030. A software reset can be executed by writing a COH to Write Register 9. The states of the SCC registers after reset are shown in Figure 3.
### Table 1. SCC Initialization Order

<table>
<thead>
<tr>
<th>Register</th>
<th>Data</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>WR9</td>
<td>1100000000</td>
<td>Hardware reset.</td>
</tr>
<tr>
<td>WR0</td>
<td>00000000XX</td>
<td>Select Shift mode (Z8030 only).</td>
</tr>
<tr>
<td>WR4</td>
<td>XXXXXXXXX</td>
<td>Transmit/Receive control. Selects Async or Sync mode.</td>
</tr>
<tr>
<td>WR1</td>
<td>0 XXXX00000</td>
<td>Select W/REQ (optional).</td>
</tr>
<tr>
<td>WR2</td>
<td>XXXXXXXXX</td>
<td>Program interrupt vector (optional).</td>
</tr>
<tr>
<td>WR3</td>
<td>XXXXXXXX0</td>
<td>Selects receiver control. Bit D0 (Rx enable) must be set to 0 at this time.</td>
</tr>
<tr>
<td>WR5</td>
<td>XXXXXXXX0X</td>
<td>Selects transmit control. Bit D3 (Tx enable) must be set to 0 at this time.</td>
</tr>
<tr>
<td>WR6</td>
<td>XXXXXXXXX</td>
<td>Program sync characters.</td>
</tr>
<tr>
<td>WR7</td>
<td>XXXXXXXXX</td>
<td>Program sync characters.</td>
</tr>
<tr>
<td>WR9</td>
<td>00000XX X</td>
<td>Select interrupt control. Bit D3 (Master interrupt enable) must be set to 0.</td>
</tr>
<tr>
<td>WR10</td>
<td>XXXXXXXXX</td>
<td>Miscellaneous control (optional).</td>
</tr>
<tr>
<td>WR11</td>
<td>XXXXXXXX</td>
<td>Clock control.</td>
</tr>
<tr>
<td>WR12</td>
<td>XXXXXXXXX</td>
<td>Time constant lower byte (optional).</td>
</tr>
<tr>
<td>WR13</td>
<td>XXXXXXXX</td>
<td>Time constant upper byte (optional).</td>
</tr>
<tr>
<td>WR14</td>
<td>XXXXXXXX0</td>
<td>Miscellaneous control. Bit D0 (BR Generator enable) must be set to 0 at this time.</td>
</tr>
<tr>
<td>WR14</td>
<td>XXXXXXXX</td>
<td>This register may require multiple writes if more than one command is used.</td>
</tr>
</tbody>
</table>

#### Stage 1. Modes and Constants

- WR3: SSSSSSSS1 Set D0 (Rx Enable).
- WR5: SSS1SSS Set D3 (Tx Enable).
- WR0: 10000000 Reset TxCRC.
- WR14: 000SSSSS1 BR Generator enable. Set bit D0 (BR Generator Enable). Enable DPLL.
- WR1: XSS00S00 Set D7, (DMA enable) if required.

#### Stage 2. Enables

- WR15: XXXXXXXX Enable external interrupts.
- WR0: 00010000 Reset EXT/STATUS twice.
- WR0: 00010000 Reset EXT/STATUS twice.
- WR1: SSSXXSXX Enable receive, transmit, and external interrupt master.
- WR9: 000SSSS Enable Master Interrupt bit D3.

---

1 (Set to one)  
0 (Set to zero)  
X (User choice)  
S (Same as previously programmed)
Label of SCC Table: ________________  SCC Base Address: ________________

Description: ________________________________________________________________

<table>
<thead>
<tr>
<th>Modes</th>
<th>Register</th>
<th>Hex</th>
<th>Binary</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>WR9</td>
<td>C</td>
<td>0</td>
<td>D7 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>WRO</td>
<td>0</td>
<td></td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>WR4</td>
<td></td>
<td></td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>WR1</td>
<td></td>
<td></td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>WR2</td>
<td></td>
<td></td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>WR3</td>
<td></td>
<td></td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>WR5</td>
<td></td>
<td></td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>WR6</td>
<td></td>
<td></td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>WR7</td>
<td></td>
<td></td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>WR9</td>
<td></td>
<td></td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>WR10</td>
<td></td>
<td></td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>WR11</td>
<td></td>
<td></td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>WR12</td>
<td></td>
<td></td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>WR13</td>
<td></td>
<td></td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>WR14</td>
<td></td>
<td></td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>WR14</td>
<td></td>
<td></td>
<td>0 0 0 0 0 0</td>
</tr>
<tr>
<td>Enables</td>
<td>WR3</td>
<td></td>
<td></td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td></td>
<td>WR5</td>
<td></td>
<td></td>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td></td>
<td>WRO</td>
<td>8</td>
<td>0</td>
<td>1 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>WR14</td>
<td></td>
<td></td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>WR1</td>
<td></td>
<td></td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>Interrupt</td>
<td>WR15</td>
<td></td>
<td></td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>WRO</td>
<td>1</td>
<td>0</td>
<td>0 0 0 1 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>WRO</td>
<td>1</td>
<td>0</td>
<td>0 0 0 1 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>WR1</td>
<td></td>
<td></td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>WR9</td>
<td></td>
<td></td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

Figure 2. SCC Initialization Worksheet
### HARDWARE RESET

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</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>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### CHANNEL RESET

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</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></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</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>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<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>
</tr>
</tbody>
</table>

**Data (.) are indeterminate, and may be a 1 or a 0.**

### Figure 3. Register Values After Reset

**INITIALIZATION EXAMPLE**

The program example in Appendix A shows how the Z8000 initializes the Z-SCC for asynchronous communication. The initialization sequence is stored in a table beginning with the program label SCCTABLE and is used by a subroutine called ZINIT. The same subroutine can use different initialization tables. The table in the program example requires two bytes for each register; the first byte is the register address and the second byte is the data. The ZINIT subroutine takes the data in this table and writes it to the SCC.

Three arguments must be set before calling the subroutine:

- The peripheral base address (in R1).
- The address of the beginning of the initialization routine (in R2).
- The number of entries in the table (in R3).

For the Z8000 to use vectored interrupts, the peripherals must be connected to AD0-AD7 of the CPU's Address/Data bus.
Appendix A. Z8000 Program Example

plzasm 1.3
LOC OBJ CODE STMT SOURCE STATEMENT

1 SCC_INIT MODULE
$liston $tty
CONSTANT


!*******************************************************!
SCC_BASE_ADDRESS

The SCC is I/O mapped at address location !FEO0. This is accomplished in hardware by decoding !chip enable (CE) from addresses AD8-AD15 and the status!lines ST0-ST3. The SCC address is assigned to the !label SCCBASE in the following equate statement. !*******************************************************!

SCCBASE := %FEO0 !Z-SCC base address !


!*******************************************************!
SCC_REGISTERS

For clarity, the address of the internal registers !is assigned a label as shown below in the equate !statements. The peripheral's ADO-AD7 pins must be !connected to the CPU's ADO-AD7 pins because the !CPU reads the interrupt vector from the low-order byte !((ADO-AD7) during an Interrupt Acknowledge cycle. !To access the peripheral's internal registers, the !least significant address bit (AO) in the register !addresses must be set to 1, and the Shift Left mode !must be selected.

WR0B := %01; WR0A := %21
WR1B := %03; WR1A := %23
WR2B := %05; WR2A := %25
WR3B := %07; WR3A := %27
WR4B := %09; WR4A := %29
WR5B := %0B; WR5A := %2B
WR6B := %0D; WR6A := %2D
WR7B := %0F; WR7A := %2F
WR8B := %11; WR8A := %31
WR9B := %13; WR9A := %33
WR10B := %15; WR10A := %35
WR11B := %17; WR11A := %37
WR12B := %19; WR12A := %39
WR13B := %1B; WR13A := %3B
WR14B := %1D; WR14A := %3D
WR15B := %1F; WR15A := %3F
GLOBAL MAIN PROCEDURE

 To initialize the SCC, the following four instructions must be included in the main program. The first three instructions load arguments into registers R1-R3 for use by the initialization subroutine ZINIT. The fourth instruction calls the ZINIT subroutine.

ENTRY

GLOBAL ZINIT PROCEDURE

 This routine is called from the main program to initialize a Z-BUS peripheral in a Z8000 system. The following arguments must be set:

ENTRY

4-181
SCC INITIALIZATION TABLE

This table is used to initialize the SCC for Asynchronous operation, 8 bits/character, 2 stop bits, no parity, x16 clock, and 9600 baud.

SCCTABLE:

<table>
<thead>
<tr>
<th>MODES AND CONSTANTS!</th>
<th>BVAL</th>
<th>WR9A</th>
</tr>
</thead>
<tbody>
<tr>
<td>001C 33</td>
<td>BVAL</td>
<td>WR9A</td>
</tr>
<tr>
<td>001D C0</td>
<td>BVAL</td>
<td>%C0</td>
</tr>
<tr>
<td>001E 29</td>
<td>BVAL</td>
<td>WR4A</td>
</tr>
<tr>
<td>001F 4C</td>
<td>BVAL</td>
<td>%4C</td>
</tr>
<tr>
<td>0020 25</td>
<td>BVAL</td>
<td>WR2A</td>
</tr>
<tr>
<td>0021 10</td>
<td>BVAL</td>
<td>%10</td>
</tr>
<tr>
<td>0022 27</td>
<td>BVAL</td>
<td>WR3A</td>
</tr>
<tr>
<td>0023 C0</td>
<td>BVAL</td>
<td>%C0</td>
</tr>
<tr>
<td>0024 2B</td>
<td>BVAL</td>
<td>WR5A</td>
</tr>
<tr>
<td>0025 E2</td>
<td>BVAL</td>
<td>%E2</td>
</tr>
<tr>
<td>0026 2D</td>
<td>BVAL</td>
<td>WR6A</td>
</tr>
<tr>
<td>0027 00</td>
<td>BVAL</td>
<td>%0</td>
</tr>
<tr>
<td>0028 2F</td>
<td>BVAL</td>
<td>WR7A</td>
</tr>
<tr>
<td>0029 00</td>
<td>BVAL</td>
<td>%0</td>
</tr>
<tr>
<td>002A 33</td>
<td>BVAL</td>
<td>WR9A</td>
</tr>
<tr>
<td>002B 01</td>
<td>BVAL</td>
<td>%01</td>
</tr>
<tr>
<td>002C 35</td>
<td>BVAL</td>
<td>WR10A</td>
</tr>
<tr>
<td>002D 00</td>
<td>BVAL</td>
<td>%0</td>
</tr>
<tr>
<td>002E 37</td>
<td>BVAL</td>
<td>WR11A</td>
</tr>
<tr>
<td>002F 56</td>
<td>BVAL</td>
<td>%56</td>
</tr>
<tr>
<td>0030 39</td>
<td>BVAL</td>
<td>WR12A</td>
</tr>
<tr>
<td>0031 06</td>
<td>BVAL</td>
<td>%06</td>
</tr>
<tr>
<td>0032 3B</td>
<td>BVAL</td>
<td>WR13A</td>
</tr>
<tr>
<td>0033 00</td>
<td>BVAL</td>
<td>%0</td>
</tr>
<tr>
<td>0034 3D</td>
<td>BVAL</td>
<td>WR14A</td>
</tr>
<tr>
<td>0035 02</td>
<td>BVAL</td>
<td>%02</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ENABLES!</th>
<th>BVAL</th>
<th>WR14A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0036 3D</td>
<td>BVAL</td>
<td>%3</td>
</tr>
<tr>
<td>0037 03</td>
<td>BVAL</td>
<td>%03</td>
</tr>
<tr>
<td>0038 27</td>
<td>BVAL</td>
<td>WR3A</td>
</tr>
<tr>
<td>0039 C1</td>
<td>BVAL</td>
<td>%C1</td>
</tr>
<tr>
<td>003A 2B</td>
<td>BVAL</td>
<td>WR5A</td>
</tr>
<tr>
<td>003B EA</td>
<td>BVAL</td>
<td>%EA</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ENABLE INTERRUPTS!</th>
<th>BVAL</th>
<th>WR15A</th>
</tr>
</thead>
<tbody>
<tr>
<td>003C 3F</td>
<td>BVAL</td>
<td>%0</td>
</tr>
<tr>
<td>003D 00</td>
<td>BVAL</td>
<td>%0</td>
</tr>
<tr>
<td>003E 21</td>
<td>BVAL</td>
<td>%10</td>
</tr>
<tr>
<td>003F 10</td>
<td>BVAL</td>
<td>%10</td>
</tr>
<tr>
<td>0040 21</td>
<td>BVAL</td>
<td>%10</td>
</tr>
<tr>
<td>0041 10</td>
<td>BVAL</td>
<td>%10</td>
</tr>
<tr>
<td>0042 33</td>
<td>BVAL</td>
<td>%09</td>
</tr>
<tr>
<td>0043 09</td>
<td>BVAL</td>
<td>%09</td>
</tr>
<tr>
<td>0044 23</td>
<td>BVAL</td>
<td>%10</td>
</tr>
<tr>
<td>0045 10</td>
<td>BVAL</td>
<td>%10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCCCOUNT:</th>
<th>BVAL</th>
<th>(((-SCCTABLE)/2)-1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0046 0015</td>
<td>WVAL</td>
<td>(((-SCCTABLE)/2)-1)</td>
</tr>
</tbody>
</table>

END ZINIT
END SCC_INIT
INTRODUCTION

The Z8038 Z-FIO is an intelligent 128x8 FIFO buffer that can link two CPUs or a CPU and a peripheral device. The Z-FIO manages data transfers by assuming Z-BUS, non-Z-BUS (a generalized microprocessor interface), 2-Wire Handshake, and 3-Wire Handshake operating modes. These modes facilitate interfacing dissimilar CPUs, or CPUs and peripherals running under differing speeds or protocols, allowing asynchronous communication and reducing I/O overhead. The width of the buffer can be expanded by connecting multiple Z-FIDs in parallel, and the depth can be expanded by using Z8060 FIFO buffers.

This application note illustrates the use of the Z-FIO in a simple data acquisition application, in which a peripheral device transfers data to a Z8002-based system at a constant rate of one byte every 100 µs. In this application, it is desirable for the system to record each byte in memory as well as dynamically keep track of the frequency of a certain data pattern. The Z-FIO facilitates this task by allowing the CPU to handle the data in blocks rather than requiring it to service an interrupt every 100 µs.

For a more complete understanding, this application note should be read in conjunction with the Z-FIO Technical Manual (Document #00-2031-01).

HARDWARE CONFIGURATION

In this application, the Port 1 side of the Z-FIO is connected to the lower byte of the system bus. The Z-BUS Low Byte mode is programmed by connecting M0 and M1 to ground. The Port 2 side receives data from the peripheral device using the Interlocked 2-Wire Handshake mode. Figure 1 shows the Z8038 hardware configuration, and Table 1 gives a description of each signal used in the application.

INITIALIZING THE Z-FIO

Before writing the initialization software, the user should keep in mind that the Z-FIO is connected to the lower byte of the system bus, so all of its registers have odd addresses. Since the least significant address bit, A0, must always equal 1 when performing byte-oriented accesses to the Z-FIO, this bit cannot be used to select registers. It is for this reason that the Right Justified Address (RJA) bit in Control Register 0 (CRO) must be reset to 0, requiring the address to be left-shifted by one bit (i.e. bits A4 - A1 are used to select the registers).

The first step in initializing the Z-FIO is the software reset, performed by writing a 1 to the Reset bit in CRO. Since no hardware reset circuit is employed, it must be assumed that the RJA bit is in an unknown state upon power-up. The first access must be performed with A4 - A0 = 00000 so that CRO is addressed regardless of the state of the RJA bit. A word-oriented output instruction (OUT) is executed, with the Z-FIO's even base address as the destination. This procedure is detailed in the program listing in the Appendix.

The ZINIT procedure completes initialization. It is called with the Z-FIO's base address in R1, and it uses the information in the table TAB to load the Z-FIO's registers. TAB is a string of byte value pairs, each pair consisting of a target register address offset and a value to be loaded into the corresponding target register. For example, the first two byte values are 01 and 00. ZINIT loads the value 00 to the target register with address offset 01.
Figure 1. Z8038 Hardware Configuration
<table>
<thead>
<tr>
<th>Signal Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AD0 - AD7 (Address/Data)</strong></td>
<td>Multiplexed, bidirectional Address/Data lines, Z-BUS compatible.</td>
</tr>
<tr>
<td><strong>DMASTB (Direct Memory Access Strobe)</strong></td>
<td>Input, active Low, tied High in this example.</td>
</tr>
<tr>
<td><strong>DS (Data Strobe)</strong></td>
<td>Input, active Low; provides timing for data transfer to or from Z-FIO.</td>
</tr>
<tr>
<td><strong>R/W (Read/Write)</strong></td>
<td>Input, active High signals CPU read from Z-FIO; active Low signals write to Z-FIO.</td>
</tr>
<tr>
<td><strong>CS (Chip Select)</strong></td>
<td>Input, active Low. Enables Z-FIO; latched on the rising edge of A5.</td>
</tr>
<tr>
<td><strong>A5 (Address Strobe)</strong></td>
<td>Input, active Low. Addresses, CS and INTACK sampled while A5 Low.</td>
</tr>
<tr>
<td><strong>INTACK (Interrupt Acknowledge)</strong></td>
<td>Input, active Low. Acknowledges an interrupt. Latched on the rising edge of A5.</td>
</tr>
<tr>
<td><strong>IEO (Interrupt Enable Out)</strong></td>
<td>Output, active High. Sends interrupt enable to lower priority device IEO pin.</td>
</tr>
<tr>
<td><strong>ICI (Interrupt Enable In)</strong></td>
<td>Input, active High. Receives interrupt enable from higher priority device IEO pin.</td>
</tr>
<tr>
<td><strong>INT (Interrupt)</strong></td>
<td>Output, open drain, active Low. Signals Z-FIO interrupt request to CPU.</td>
</tr>
</tbody>
</table>

**2-Wire Handshake: Port 2 Side**

<table>
<thead>
<tr>
<th>Signal Description</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>D0 - D7 (Data)</strong></td>
<td>Bidirectional data bus. Input in this example.</td>
</tr>
<tr>
<td><strong>RFD/DAV (Ready for Data/Data Available)</strong></td>
<td>Output, RFD active High. While port is input, signals that Z-FIO is ready to receive data.</td>
</tr>
<tr>
<td><strong>ACKIN (Acknowledgement Input)</strong></td>
<td>Input, active Low. Signals that input data is valid. Pull-up resistor ensures that ACKIN is High when handshake is enabled.</td>
</tr>
<tr>
<td><strong>FULL</strong></td>
<td>Output, input, open drain, active High. Must be pulled High in this example since the conditions for setting the Full Interrupt Pending (IP) bit are: Buffer is full, and FULL input is High.</td>
</tr>
<tr>
<td><strong>EMPTY</strong></td>
<td>Output, input, open drain, active High. Must be pulled High in this example since the conditions for setting the Empty IP bit are: Buffer is empty, and EMPTY input is High.</td>
</tr>
</tbody>
</table>
INTERRUPT CONSIDERATIONS

Essential to this application are the powerful vectored interrupt capabilities inherent in Z-BUS architecture. When the Z8002 VI input is pulled low, a vectored interrupt is requested. If the Vectored Interrupt Enable (VIE) bit in the Flag Control Word (FCW) is set to 1, the Z8002 executes an Interrupt Acknowledge cycle during which it reads a vector from the lower byte of the Address/Data bus. The Z8002 then loads the Program Status registers (which include the FCW and the PC) from the vector table in the Program Status Area.

The Z-F10 interrupts the CPU each time the buffer is full. In servicing the Buffer Full interrupt, the CPU performs the necessary overhead operations and then executes an Input Increment and Repeat Byte (INIRB) instruction to move the data from the Z-F10 to memory.

In order to dynamically count the occurrences of a certain data pattern, the Z-F10 must interrupt the INIRB instruction each time the pattern appears in the Data Buffer register. (INIRB is an iterative instruction and can be interrupted after each execution of the basic operation.) Finally, when the buffer is empty, the Z-F10 interrupts the INIRB instruction again so that a 1 can be loaded into the iteration counter (in this case RO) and the block move can be terminated. This method of inputting data until the Z-F10 is empty is more efficient than inputting a fixed number of bytes, because the block size varies according to the amount of time spent servicing Pattern Match interrupts.

Initializing the Vector Table

The vector table in the Program Status Area consists of an FCW, which is used for all vectored interrupts, and up to 256 word values that can be loaded into the CPU's PC during a Vectored Interrupt Acknowledge cycle. These values correspond to the 256 possible values of the Interrupt Vector that is read on the lower byte of the Address/Data bus. The vector value 0 selects the first PC value, the vector value 1 selects the second PC value, and so on up to the vector value 255.

Though Port 1 has only one Interrupt Vector register, the three interrupt conditions used in this application (Buffer Empty, Buffer Full, and Pattern Match) can generate unique vectors via the Vector Includes Status feature. This feature encodes the interrupt status into bits D1 - D3 of the vector according to the convention shown in Figure 2. Assuming a base vector value of 00H, Table 2 gives the vectors that the interrupt conditions generate, their corresponding PC values, and the byte offsets that address these values in the Program Status Area.

<table>
<thead>
<tr>
<th>Interrupt Condition</th>
<th>Interrupt Vector (hex)</th>
<th>PC Value</th>
<th>Byte Offset (decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer Empty</td>
<td>02</td>
<td>PC₃</td>
<td>34</td>
</tr>
<tr>
<td>Buffer Full</td>
<td>04</td>
<td>PC₅</td>
<td>38</td>
</tr>
<tr>
<td>Pattern Match</td>
<td>0A</td>
<td>PC₁₁</td>
<td>50</td>
</tr>
</tbody>
</table>

The software routines show how these byte offsets (in conjunction with the PSAP) form indexed addresses to initialize the vector table.

Buffer Full Interrupt

Buffer Full is the only interrupt that interrupts the background task. Since one byte of data is moved to the buffer every 100 μs, it takes 128 x 100 = 12.8 μs from the time the buffer is empty until the Buffer Full condition requires service. The primary task of the FULL service routine is to execute the INIRB instruction, which moves the data from the Z-F10 to a memory buffer starting at location BUF (6000H). Before INIRB is executed, the Pattern Match interrupt is enabled, the Full interrupt is disabled, and the Disable Lower Chain command is issued so that no interrupt sources of lower priority than the Z-F10 can interrupt the FULL routine.
After execution of the INIRB instruction, the destination pointer (R1) is decremented to compensate for the extra iteration that takes place after the buffer goes empty. The Clear Full Interrupt Pending command is issued in case the Full IP bit has been set since the most recent Clear Full IP command (e.g. the peripheral device transferred a byte to the buffer just after the first iteration of the INIRB instruction, thus causing the buffer to go full and the Full IP bit to be set). The Full IE bit is then set so the Z-FIO can cause an interrupt the next time it is full, and the Pattern Match IE bit is cleared to prevent a Pattern Match condition from interrupting the background task. Finally, the lower daisy chain is enabled and control is returned to the background task.

**Buffer Empty Interrupt**

The Buffer Empty IP bit is set whenever the Z-FIO makes a transition from a "not-empty" state to an empty state. In this application, it is set when the INIRB instruction reads the last byte from the Z-FIO buffer. Since the Buffer Empty interrupt has lower priority than the Buffer Full interrupt, the Full Interrupt Under Service (IUS) bit must be cleared if the Buffer Empty condition is to preempt the FULL service routine. (Z-BUS interrupt sources hold their Interrupt Enable Output (IEO) line Low whenever their IUS bit is set.) The EMPTY service routine loads a 1 into the iteration counter (RO), causing the INIRB instruction to be terminated after the next iteration. The service routine then clears the Empty IP and IUS bits and returns control to the FULL routine.

**Pattern Match Interrupt**

The Pattern Match interrupt is a higher priority interrupt than the Buffer Full interrupt, and it can preempt the FULL routine if the Pattern Match IE bit is set. The Pattern Match IP bit is set whenever the Data Buffer register contains the pattern (specified as 55_H by the initialization sequence). The PAT service routine simply increments the pattern counter (RL3), clears the Pattern Match IP and IUS bits, and returns control to the FULL routine. The IP and IUS bits are cleared in separate commands to prevent a spurious interrupt caused by IUS being cleared before IP is cleared. The background task can interpret the value in RL3 as the number of times the pattern 55_H appears in the most recently transferred block of data.

**APPENDIX**

Following is a listing of the software used in this application. It is assumed that the PSAP has been initialized and that the Z8002 is in System mode when it enters the MAIN procedure. The background task is simulated by the "JR $" instruction.

Under ZINIT, each address offset shown is keyed to the name of the corresponding register, and each loaded value is keyed to the effect of the load.
1 RECEIVE MODULE
2 EXTERNAL ZINIT PROCEDURE
3 INTERNAL CONSTANT
4 BUF := %6000 !MEMORY BUFFER!
5 FIOPA := %FD00 !FIO BASE ADDR!
6 FDATA := %FD1F !FIO DATA REG!
7 CRO := %FD01 !CONTROL REG 0!
8 ISR1 := %FD07 !INTR STATUS REG 1!
9 ISR3 := %FD0B !INTR STATUS REG 3!
0
0000 11 GLOBAL MAIN PROCEDURE
12 ENTRY
13
0000 7C01 14 DI VI !DISABLE VECTORED INTR!
15
16 !INITIALIZE FIO!
0002 BD01 17 LDK RO,#1
0004 3B06 FD00 18 OUT FIOPA,RO !RESET FIO WITH EVEN ADDR!
0008 2101 FD00 19 LD R1,#FIOPA
000C 5F00 0000 20 CALL ZINIT
21
22 !INITIALIZE VECTOR TABLE!
0010 7D15 23 LDCTL R1,PSAP !LOAD PROG STATUS AREA PTR!
0012 4D15 001C 24 LD 28(R1),#%4000 !LOAD FCW FOR VECTORED INTR!
0016 4000
0018 7602 0038' 25 LDA R2,FULL !LOAD ADDR OF FULL PROCEDURE!
001C 6F12 0026 26 LD 38(R1),R2 !ENTER ADDR IN VECTOR TABLE!
0020 7602 0084' 27 LDA R2,PAT !ENTER ADDR OF PAT PROCEDURE!
0024 6F12 0032 28 LD 50(R1),R2 !ENTER ADDR IN VECTOR TABLE!
0028 7602 007A' 29 LDA R2,EMPTY !LOAD ADDR OF EMPTY PROCEDURE!
002C 6F12 0022 30 LD 34(R1),R2 !ENTER ADDR IN VECTOR TABLE!
31
32
0030 2101 6000 33 LD R1,#BUF !LOAD ADDR OF MEMORY BUFFER!
0034 7C05 34 EI VI !ENABLE VECTORED INTR!
0036 E8FF 35 JR $ !BACKGROUND TASK!

0038 36 END MAIN
37
0038 38 INTERNAL FULL PROCEDURE
39 ENTRY
40
<table>
<thead>
<tr>
<th>LOC</th>
<th>OBJ CODE</th>
<th>STMT SOURCE STATEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0038</td>
<td>2100 O0C0</td>
<td>LD RO,#%O0C0</td>
</tr>
<tr>
<td>003C</td>
<td>3A06 FD07</td>
<td>OUTB ISR1,RH0 !SET PATTERN MATCH IE!</td>
</tr>
<tr>
<td>0040</td>
<td>3A06 FD01</td>
<td>OUTB CRO,RLO !DISABLE LOWER DAISY CHAIN!</td>
</tr>
<tr>
<td>0044</td>
<td>2100 20E0</td>
<td>LD RO,#%20E0</td>
</tr>
<tr>
<td>0048</td>
<td>3A06 FD0B</td>
<td>OUTB ISR3,RH0 !CLEAR FULL IP &amp; IUS!</td>
</tr>
<tr>
<td>004C</td>
<td>3A06 FD0B</td>
<td>OUTB ISR3,RLO !CLEAR FULL IE!</td>
</tr>
<tr>
<td>0050</td>
<td>8C88</td>
<td>CLRB RL3 !INITIALIZE COUNT!</td>
</tr>
<tr>
<td>0052</td>
<td>2102 FD1F</td>
<td>LD R2,#FDATA</td>
</tr>
<tr>
<td>0056</td>
<td>7C05</td>
<td>EI VI !ENABLE VECTORED INTR!</td>
</tr>
<tr>
<td>0058</td>
<td>3A20 0010</td>
<td>INIRB @R1,&amp;R2,RO !READ DATA FROM FIO!</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>005C</td>
<td>7C01</td>
<td>DI VI !DISABLE VECTORED INTR!</td>
</tr>
<tr>
<td>005E</td>
<td>AB10</td>
<td>DEC R1</td>
</tr>
<tr>
<td>0060</td>
<td>2100 A0C0</td>
<td>LD RO,#%A0C0</td>
</tr>
<tr>
<td>0064</td>
<td>3A06 FD0B</td>
<td>OUTB ISR3,RH0 !CLEAR FULL IP!</td>
</tr>
<tr>
<td>0068</td>
<td>3A06 FD0B</td>
<td>OUTB ISR3,RLO !SET FULL IE!</td>
</tr>
<tr>
<td>006C</td>
<td>2100 0E9C</td>
<td>LD RO,#%0E9C</td>
</tr>
<tr>
<td>0070</td>
<td>3A06 FD07</td>
<td>OUTB ISR1,RH0 !CLEAR PATTERN MATCH IE!</td>
</tr>
<tr>
<td>0074</td>
<td>3A06 FD01</td>
<td>OUTB CRO, RLO !ENABLE LOWER DAISY CHAIN!</td>
</tr>
<tr>
<td>0078</td>
<td>7B00</td>
<td>INT IRET</td>
</tr>
<tr>
<td>007A</td>
<td></td>
<td>END FULL</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>007A</td>
<td></td>
<td>64 INTERNAL EMPTY PROCEDURE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>65 ENTRY</td>
</tr>
<tr>
<td>007A</td>
<td>BD01</td>
<td>LDK RO,#1 !TERMINATE BLOCK MOVE!</td>
</tr>
<tr>
<td>007C</td>
<td>C302</td>
<td>LDB RH3,#%02</td>
</tr>
<tr>
<td>007E</td>
<td>3A36 FD0B</td>
<td>OUTB ISR3,RH3 !CLEAR EMPTY IP AND IUS!</td>
</tr>
<tr>
<td>0082</td>
<td>7B00</td>
<td>INT IRET</td>
</tr>
<tr>
<td>0084</td>
<td></td>
<td>70 END EMPTY</td>
</tr>
<tr>
<td></td>
<td></td>
<td>71</td>
</tr>
<tr>
<td>0084</td>
<td></td>
<td>72 INTERNAL PAT PROCEDURE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>73 ENTRY</td>
</tr>
<tr>
<td>0084</td>
<td>A880</td>
<td>INCB RL3 !INCREMENT COUNT!</td>
</tr>
<tr>
<td>0086</td>
<td>2104 OA06</td>
<td>LD R4,#%OA06</td>
</tr>
<tr>
<td>008A</td>
<td>3A46 FD07</td>
<td>OUTB ISR1,RH4 !CLEAR PATTERN MATCH IP!</td>
</tr>
<tr>
<td>008E</td>
<td>3AC6 FD07</td>
<td>OUTB ISR1,RL4 !CLEAR PATTERN MATCH IUS!</td>
</tr>
<tr>
<td>0092</td>
<td>7B00</td>
<td>INT IRET</td>
</tr>
<tr>
<td>0094</td>
<td></td>
<td>79 END PAT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>80 END RECEIVE</td>
</tr>
</tbody>
</table>

1
2 ZIN MODULE

0000
3 GLOBAL ZINIT PROCEDURE

4

5 ! THIS IS A GENERAL ROUTINE USED !
6 ! TO INITIALIZE A Z-BUS PERIPHERAL !
7 ! IN THIS EXAMPLE IT INITIALIZES !
8 ! THE Z-FIO. !
9 !

4-189
LOC  OBJ CODE  SIMT SOURCE STATEMENT

10  ! R1 = PERIPHERAL BASE ADDR    !
11  ! R2 = ADDR OF TABLE       !
12  ! R3 = NO. OF BYTES TO BE OUTPUT !
13
14  ENTRY
0000 7602 0014'  15  LDA R2,TAB
0004 6103 0024'  16  LD R3,COUNT
17  LOOP:
0008 2029  18  LDB RL1,R2
000A A920  19  INC R2
000C 3A22 0318  20  OUTIB @R1,R2,R3
21
0010 ECFB  22  JR NOV,LOOP
0012 9E08  23  RET
24
25  TAB:
0014 01  26  BVAL %01 !CONTROL REGISTER 0!
0015 00  27  BVAL %00 !CLEAR RESET!
0016 01  28  BVAL %01 !CONTROL REGISTER 0!
0017 0C  29  BVAL %0C !INTERLOCKED HS PORT!
0018 15  30  BVAL %15 !CONTROL REGISTER 3!
0019 50  31  BVAL %50 !INPUT TO CPU!
001A 13  32  BVAL %13 !CONTROL REGISTER 2!
001B 03  33  BVAL %03 !ENABLE PORT 2!
001C 1B  34  BVAL %1B !PATTERN MATCH REGISTER!
001D 55  35  BVAL %55 !PATTERN IS 55!
001E 0B  36  BVAL %0B !INTERRUPT STATUS REGISTER 3!
001F CC  37  BVAL %CC !SET FULL AND EMPTY IE!
0020 01  38  BVAL %01 !CONTROL REGISTER 0!
0021 9C  39  BVAL %9C !SET MIE BIT!
40
41  COUNT:
0022 0008  42  WVAL ((-TAB)/2 -1)
0024  43  END ZINIT
44  END ZIN
Zilog Sales Offices and Technical Centers

West
Sales & Technical Center
Zilog, Incorporated
1315 Dell Avenue
Campbell, CA 95008
Phone: (408) 370-8120
TWX: 910-338-7621

Sales & Technical Center
Zilog, Incorporated
18023 Sky Park Circle
Suite J
Irvine, CA 92714
Phone: (714) 549-2891
TWX: 910-95-2803

Sales & Technical Center
Zilog, Incorporated
16643 Sherman Way
Suite 430
Van Nuys, CA 91406
Phone: (213) 999-7485
TWX: 910-495-1765

South
Sales & Technical Center
Zilog, Incorporated
4851 Keller Springs Road,
Suite 211
Dallas, TX 75248
Phone: (214) 999-9090
TWX: 910-960-5850

Zilog, Incorporated
7113 Burnet Rd.
Suite 207
Austin, TX 78757
Phone: (512) 453-3216

East
Sales & Technical Center
Zilog, Incorporated
951 North Plum Grove Road
Suite F
Schaumburg, IL 60195
Phone: (312) 885-8080
TWX: 910-291-1064

Sales & Technical Center
Zilog, Incorporated
28349 Chagrin Blvd.,
Suite 109
Woodmere, OH 44122
Phone: (216) 831-7040
FAX: 216-831-2957

Zilog, Incorporated
8028 TAUFKIRCHEN
Munich, West Germany
Phone: 89-612-6046
Telex: 529110 Zilog d.

United Kingdom
Zilog (U.K.) Limited
Zilog House
43-53 Moorbridge Road
Maidenhead
Berkshire, SL6 8PL England
Phone: 0628-39200
Telex: 848609

France
Zilog, Incorporated
Cedex 31
92098 Paris La Defense
France
Phone: (1) 334-60-09
TWX: 611455

West Germany
Zilog GmbH
Eschenstrasse 8
D-8028 TAUFKIRCHEN
Munich, West Germany
Phone: 89-612-6046
Telex: 529110 Zilog d.

Japan
Zilog Japan K.K.
Konparu Bldg. 5F
2-8 Akasaka 4-Chome
Minato-Ku, Tokyo 107
Japan
Phone: (81) (03) 587-0528
Telex: 2422024 A/B: Zilog J