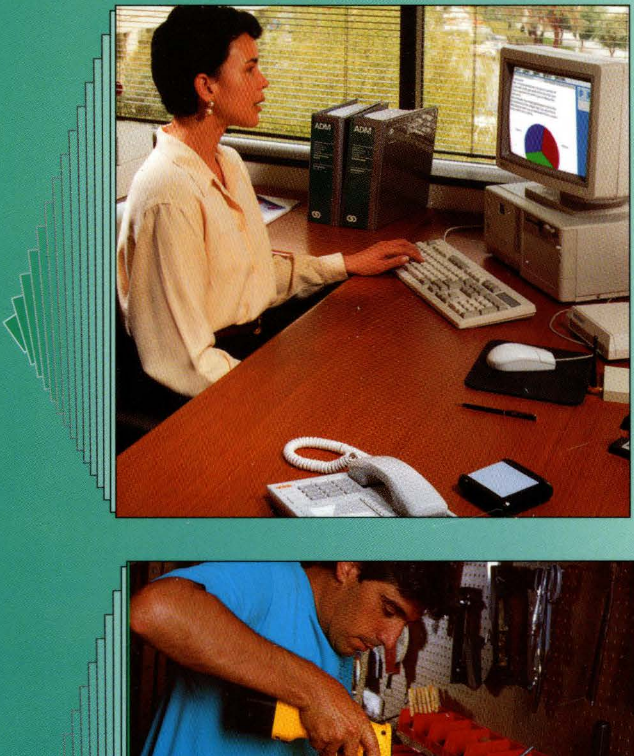
Z8® Microcontrollers



Embedded Controllers can be used in a variety of applications.

User's Manual

Z8® Microcontrollers

User's Manual

Q2/95 UM95Z80010

Overview

Zilog's Focus on Application-Specific Products Helps You Maintain Your Technological Edge

The Z8® Microcontroller User's Manual consists of the following:

- **Z8® Architecture Technical Description**
- **Zilog Software User's Guides**
  - **asm Z8® Cross Assembler**
  - **Zilog Universal Object File Utilities**
- **Zilog General Information**
  - **General Terms and Conditions**
  - **Zilog Sales Offices, Representatives, and Distributors**
  - **Zilog Literature Guide**

Application notes and other information on Zilog specialty software and documentation is available through the Zilog Bulletin Board Service (ZBBS), which can be reached by calling 408-370-8024 (up to 28.8 baud supported, 8-N-1 connections, and ANSI/BBS terminal emulation setup recommended).

Zilog's products are not authorized for use as critical components in life support devices or systems unless a specific written agreement pertaining to such intended use is executed between the customer and Zilog prior to use. Life support devices or systems are those which are intended for surgical implantation into the body, or which sustains life whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in significant injury to the user.

Zilog, Inc. 210 East Hacienda Ave.
Campbell, CA 95008-6600
Telephone (408) 370-8000
Telex 910-338-7621
FAX 408 370-8056
Internet: http://www.zilog.com/zilog

ZILOG Z8® MICROCONTROLLERS USER'S MANUAL

TABLE OF CONTENTS

I. Z8® MICROCONTROLLER TECHNICAL DESCRIPTION

CHAPTER TITLE AND SUBSECTIONS **PAGE**

CHAPTER 1: DISCRETE Z8® PRODUCT OVERVIEW

CHAPTER 2: ADDRESS SPACE

CHAPTER 3: CLOCK

CHAPTER 4: RESET—WATCH-DOG TIMER

CHAPTER 5: I/O PORTS

CHAPTER TITLE AND SUBSECTIONS **PAGE**

CHAPTER 5: I/O PORTS (CONTINUED)

CHAPTER 6: COUNTER/TIMERS

CHAPTER 7: INTERRUPTS

CHAPTER 8: POWER-DOWN MODES

CHAPTER 9: SERIAL I/O

CHAPTER TITLE AND SUBSECTIONS **PAGE**

CHAPTER 10: EXTERNAL INTERFACE

CHAPTER 11: ADDRESSING MODES

CHAPTER 12: INSTRUCTION SET

CHAPTER 13: ZILOG EMULATORS/SOFTWARE

CHAPTER TITLE AND SUBSECTIONS **PAGE**

II. ZILOG Z8® SOFTWARE

ASM Z8® CROSS ASSEMBLER USER'S GUIDE

CHAPTER TITLE AND SUBSECTIONS PAGE

CHAPTER 1: OVERVIEW

CHAPTER 2: ASSEMBLY LANGUAGE SYNTAX

CHAPTER 3: PSEUDO-OPS

CHAPTER 4: MACROS

CHAPTER 5: PROGRAM INVOCATION

APPENDICES

ZILOG UNIVERSAL OBJECT FILE UTILITIES USER'S GUIDE

CHAPTER TITLE AND SUBSECTIONS **PAGE**

Chapter Title and Subsections **Page**

Chapter 10: PROTOCOL

Chapter 11: Other Programs

Appendices

III. ADDITIONAL INFORMATION

Z8® Microcontroller Technical Description

I

Zilog Z8® Software

II

Zilog General Information

III

CHAPTER 1
DISCRETE Z8® PRODUCT OVERVIEW

1.1 Z8 MCU FAMILY OVERVIEW

The Zilog Z8® microcontroller product line continues to expand with new product introductions. Zilog MCU products are targeted for cost-sensitive, high-volume applications including consumer, automotive, security, and HVAC. It includes ROM-based products geared for high-volume production (where software is stable) and one-time programmable (OTP) equivalents for prototyping as well as volume production where time to market or code flexibility is critical (Table 1-1). A variety of packaging options are available including plastic DIP, SOIC, PLCC, and QFP.

A generalized Z8 MCU block diagram is shown in Figure 1-1. The same on-chip peripherals are used across the MCU product line with the primary differences being the amount of ROM/RAM, number of I/O lines present, and packaging/temperature ranges available. This allows code written for one MCU device to be easily ported to another family member.

1.1.1 Key Product Line Features

- **General-Purpose Register (GPR) File Architecture:** Every RAM register acts like an accumulator, speeding instruction execution and maximizing coding efficiency. Working register groups allow fast context switching.

- **Flexible I/O:** I/O byte, nibble, and/or bit programmable as inputs or outputs. Outputs are software programmable as open-drain or push-pull on a port basis. Inputs are Schmitt-triggered with auto latches to hold unused inputs at a known voltage state.

- **Analog Inputs:** Three input pins are software programmable as digital or analog inputs. When in the analog mode, two comparator inputs are provided with a common reference input. These inputs are ideal for a variety of common functions, including threshold level detection, analog-to-digital conversion, and short circuit detection. Each analog input provides a unique maskable interrupt input.

- **Timer/Counter(T/C):** The T/C consists of a programmable 6-bit prescaler and 8-bit downcounter, with maskable interrupt upon end-of-count. Software controls T/C load/start/stop, countdown read (at any time on the fly), and maskable end-of-count interrupt. Special functions available include T_{IN} (external counter input, external gate input, or external trigger input) and T_{OUT} (external access to timer output or the internal system clock.) These special functions allow accurate hardware input pulse measurement and output waveform generation.

- **Interrupts:** There are six vectored interrupt sources with software-programmable enable and priority for each of the six sources.

- **Watch-Dog Timer (WDT):** An internal WDT circuit is included as a fail-safe mechanism so that if software strays outside the bounds of normal operation, the WDT will timeout and reset the MCU. To maximize circuit robustness and reliability, the default WDT clock source is an internal RC circuit (isolated from the device clock source).

- **Auto Reset/Low-Voltage Protection:** All family devices have internal Power-On Reset. ROM devices add low-voltage protection. Low-voltage protection ensures the MCU is in a known state at all times (in active RUN mode or RESET) without external hardware (or a device reset pin).

- **Low-EMI Operation:** Mode is programmable via software or as a mask option. This new option provides for reduced radiated emission via clock and output drive circuit changes.

- **Low-Power:** CMOS with two standby modes; STOP and HALT.

- **Full Z8 Instruction Set:** Forty-eight basic instructions, supported by six addressing modes with the ability to operate on bits, nibbles, bytes, and words.

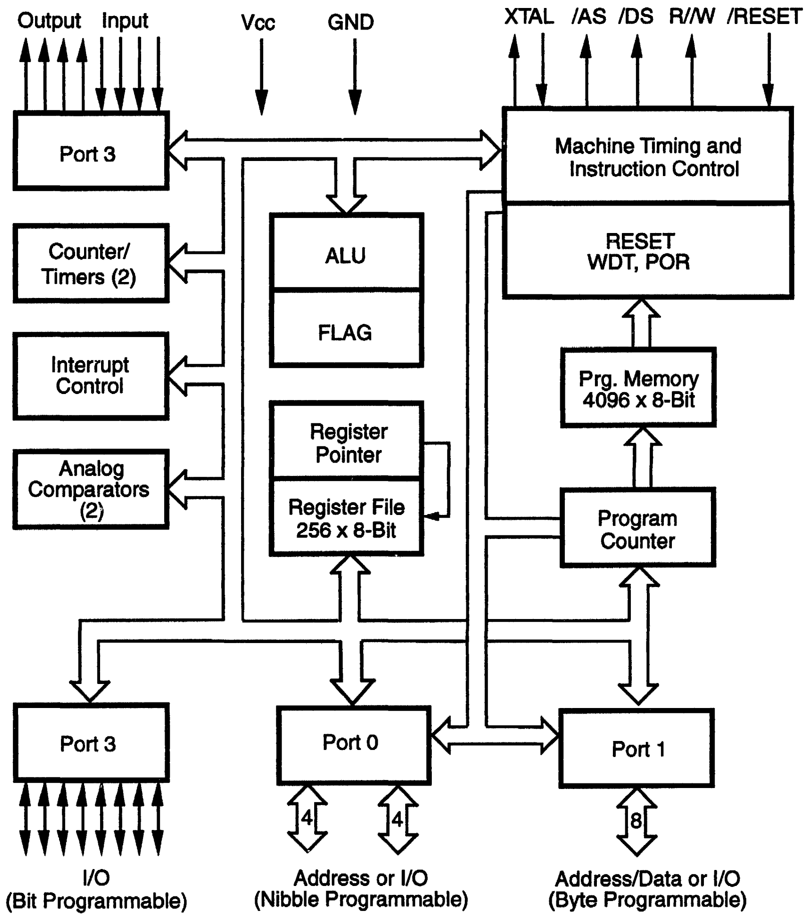


Figure 1-1. Z8® MCU Block Diagram

1.1.2 Product Development Support

The Z8® MCU product line is fully supported with a range of cross assemblers, C compilers, ICEBOX emulators, single and gang OTP/EPROM programmers, and software simulators.

The Z86CCP00ZEM low-cost Z8 CCP™ real-time emulator/programmer kit was designed specifically to support all the products outlined in Table 1-1.

Table 1-1. Zilog General-Purpose Microcontroller Product Family

PRODUCT	ROM/ RAM	I/0	T/C	AN IN	INT	WDT	POR	Vbo	RC	SPEED (MHz)	PIN COUNT
Z86C03	512/60	14	1	2	6	F	Y	Y	Y	8	18
Z86C03	512/60	14	1	2	6	F	Y	N	Y	8	18
Z86C04	1K/124	14	2	2	6	F	Y	Y	N	8	18
Z86E04	1K/124	14	2	2	6	F	Y	N	N	8	18
Z86C06	1K/124	14	2	2	6	P	Y	Y	Y	12	18
Z86E06	1K/124	14	2	2	6	P	Y	N	Y	12	18
Z86C08	2K/124	14	2	2	6	F	Y	Y	N	12	18
Z86E08	2K/124	14	2	2	6	F	Y	N	N	12	18
Z86C30	4K/236	24	2	2	6	P	Y	Y	Y	12	28
Z86E30	4K/236	24	2	2	6	P	Y	N	Y	12	28
Z86C31	2K/124	24	2	2	6	P	Y	Y	Y	8	28
Z86E31	2K/124	24	2	2	6	P	Y	N	Y	8	28
Z86C40	4K/236	32	2	2	6	P	Y	Y	Y	12	40/44
Z86E40	4K/236	32	2	2	6	P	Y	N	Y	12	40/44

Note: Z86Cxx signify ROM devices; Z86Exx signify EPROM devices; F = fixed; P = programmable.

The Z86CCP00ZEM kit comes with:

- Z8 CCP Emulator/Programmer Module
- 18-pin Target Connection Cable
- WINDOWS-based GUI Host Software
- DOS-based ZASM LINKER/LOADER
- Documentation: Z8MOBJ Linker/Loader User's Guide, Z8 Cross Assembler User's Guide, Z8 Emulator GUI User's Guide, Discrete Z8 MCU Product Specifications Databook, and Z8 MCU Technical Manual.

A Z8 CCP Emulator Accessory Kit (Z8CCP00ZAC) is also available and provides an RS-232 cable and power cable along with the 28- and 40- pin ZIF sockets and 28 and 40 pin target connector cables required to emulate/program 28/40 pin devices.

CHAPTER 2
ADDRESS SPACE

2.1 INTRODUCTION

Four address spaces are available for the Z8® microcontroller:

- The Z8 Standard Register File contains addresses for peripheral, control, all general-purpose, and all I/O port registers. This is the default register file specification.
- The Z8 Expanded Register File (ERF) contains addresses for control and data registers for additional peripherals/features.
- Z8 External Program Memory contains addresses for all memory locations having executable code and/or data.
- Z8 External Data Memory contains addresses for all memory locations that hold data only, whether internal or external.

2.2 Z8 STANDARD REGISTER FILE

The Z8 Standard Register File totals up to 256 consecutive bytes (Registers). The register file consists of 4 I/O ports (00H-03H), 236 General-Purpose Registers (04H-EFH), and 16 control registers (F0H-FFH). Table 2-1 shows the layout of the register file, including register names, locations, and identifiers.

Table 2-1. Z8 Standard Register File

Hex Address	Register Description	Register Identifier
FF	Stack Pointer Low Byte	SPL
FE	Stack Pointer High Byte	SPH
FD	Register Pointer	RP
FC	Program Control Flags	FLAGS
FB	Interrupt Mask Register	IMR
FA	Interrupt Request Register	IRQ
F9	Interrupt Priority Register	IPR
F8	Port 0-1 Mode Register	P01M
F7	Port 3 Mode Register	P3M
F6	Port 2 Mode Register	P2M
F5	T0 Prescaler	PRE0
F4	Timer/Counter 0	T0
F3	T1 Prescaler	PRE1
F2	Timer/Counter 1	T1
F1	Timer Mode	TMR
F0	Serial I/O	SIO
EF		R239
.		.
.	General-Purpose Registers (GPR)	.
.		.
04		R4
03	Port 3	P3
02	Port 2	P2
01	Port 1	P1
00	Port 0	P0

Note: Refer to the product specification to determine which registers are available for use on any specific device.

2.2 Z8 STANDARD REGISTER FILE (Continued)

Registers can be accessed as either 8-bit or 16-bit registers using Direct, Indirect, or Indexed Addressing. All 236 general-purpose registers can be referenced or modified by any instruction that accesses an 8-bit register, without the need for special instructions. Registers accessed as 16 bits are treated as even-odd register pairs (there are 118 valid pairs). In this case, the data's Most Significant Byte (MSB) is stored in the even numbered register, while the Least Significant Byte (LSB) goes into the next higher odd numbered register (Figure 2-1).

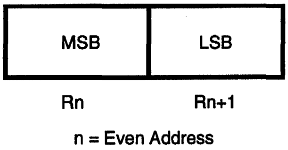


Figure 2-1. 16-Bit Register Addressing

By using a logical instruction and a mask, individual bits within registers can be accessed for bit set, bit clear, bit complement, or bit test operations. For example, the instruction AND R15, MASK performs a bit clear operation. Figure 2-2 shows this example.

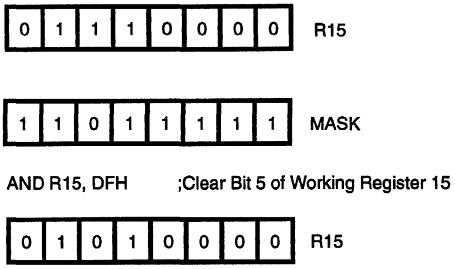


Figure 2-2. Accessing Individual Bits (Example)

When instructions are executed, registers are read when defined as sources and written when defined as destinations. All General-Purpose Registers function as accumulators, address pointers, index registers, stack areas, or scratch pad memory.

2.2.1 General-Purpose Registers

General-Purpose Registers (GPR) are undefined after the device is powered up. The registers keep their last value after any reset, as long as the reset occurs in the V_{CC} voltage-specified operating range. It will not keep its last state from a V_{LV} reset if V_{CC} drops below 1.8v.

Note: Registers in Bank E0-EF may only be accessed through the working register and indirect addressing modes. Direct access cannot be used because the 4-bit working register address mode already uses the format [E I dst], where dst represents the working register number from 0H to FH.

2.2.2 RAM Protect

The upper portion of the register file address space 80FH to EFH (excluding the control registers) may be protected from reading and writing. The RAM Protect bit option is mask-programmable and is selected by the customer when the ROM code is submitted. After the mask option is selected, the user activates this feature from the internal ROM code to turn off/on the RAM Protect by loading either a 0 or 1 into the IMR register, bit D6. A 1 in D6 enables RAM Protect. Only devices that use registers 80H to EFH offer this feature.

2.2.3 Working Register Groups

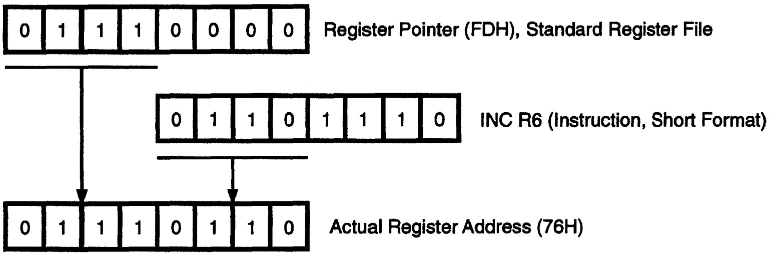
Z8® instructions can access 8-bit registers and register pairs (16-bit words) using either 4-bit or 8-bit address fields. 8-bit address fields refer to the actual address of the register. For example, Register 58H is accessed by calling upon its 8-bit binary equivalent, 01011000 (58H).

With 4-bit addressing, the register file is logically divided into 16 Working Register Groups of 16 registers each, as shown in Table 2-2. These 16 registers are known as Working Registers. A Register Pointer (one of the control registers, FDH) contains the base address of the active Working Register Group. The high nibble of the Register Pointer determines the current Working Register Group.

When accessing one of the Working Registers, the 4-bit address of the Working Register is combined within the upper four bits (high nibble) of the Register Pointer, thus forming the 8-bit actual address. Figure 2-3 illustrates this operation. Since working registers are typically specified by short format instructions, there are fewer bytes of code needed, which reduces execution time. In addition, when processing interrupts or changing tasks, the Register Pointer speeds context switching. A special Set Register Pointer (SRP) instruction sets the contents of the Register Pointer.

Table 2-2. Working Register Groups

Register Pointer (FDH) High Nibble	Working Register Group (HEX)	Actual Registers (HEX)
1111(B)	F	F0-FF
1110(B)	E	E0-EF
1101(B)	D	D0-DF
1100(B)	C	C0-CF
1011(B)	B	B0-BF
1010(B)	A	A0-AF
1001(B)	9	90-9F
1000(B)	8	80-8F
0111(B)	7	70-7F
0110(B)	6	60-6F
0101(B)	5	50-5F
0100(B)	4	40-4F
0011(B)	3	30-3F
0010(B)	2	20-2F
0001(B)	1	10-1F
0000(B)	0	00-0F



0 1 1 1 0 0 0 0
Register Pointer (FDH), Standard Register File
0 1 1 0 1 1 1 0
INC R6 (Instruction, Short Format)
0 1 1 1 0 1 1 0
Actual Register Address (76H)

Figure 2-3. Working Register Addressing Examples

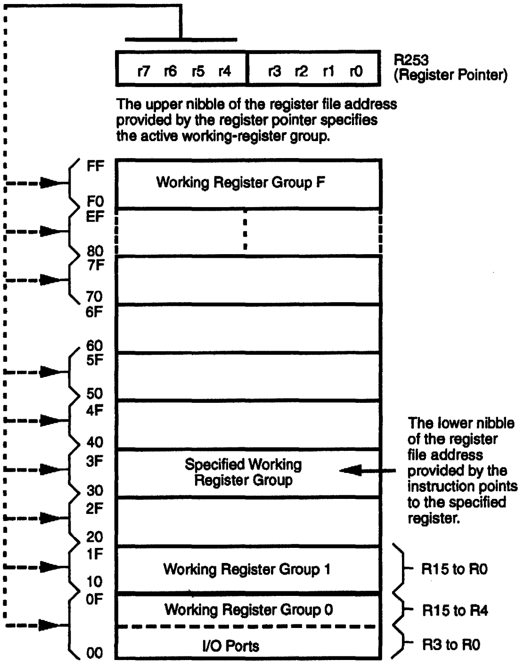

r7 r6 r5 r4 | r3 r2 r1 r0
R253
(Register Pointer)
The upper nibble of the register file address provided by the register pointer specifies the active working-register group.
FF
F0
EF
80
7F
70
6F
60
5F
50
4F
40
3F
30
2F
20
1F
10
0F
00
Working Register Group F
Specified Working Register Group
Working Register Group 1
Working Register Group 0
I/O Ports
The lower nibble of the register file address provided by the instruction points to the specified register.
R15 to R0
R15 to R4
R3 to R0

Figure 2-4. Register Pointer

Note: The full register file is shown. Please refer to the selected device product specification for actual file size.

2.2.4 Error Conditions

Registers in the Z8® Standard Register File must be correctly used because certain conditions produce inconsistent results and should be avoided.

- Registers F3H and F5H-F9H are write-only registers. If an attempt is made to read these registers, FFH is returned. Reading any write-only register will return FFH.

- When register FDH (Register Pointer) is read, the least significant four bits (lower nibble) will indicate the current Expanded Register File Bank. (Example: 0000 indicates the Standard Register File, while 1010 indicates Expanded Register File Bank A.)

- When Ports 0 and 1 are defined as address outputs, registers 00H and 01H will return 1s in each address bit location when read.

- Writing to bits that are defined as timer output, serial output, or handshake output will have no effect.

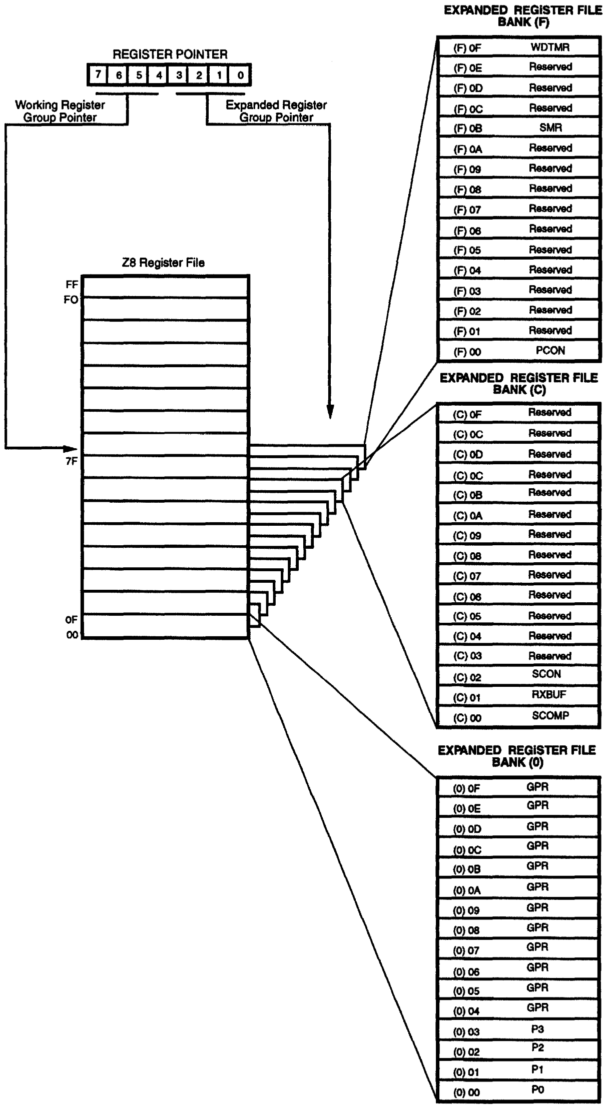
- The Z8 instruction DJNZ uses any general-purpose working register as a counter.

- Logical instructions such as OR and AND require that the current contents of the operand be read. They therefore will not function properly on write-only registers.

- The WDTMR register must be written within the first 64 internal system clocks (SCLK) of operation after a reset.

2.3 Z8 EXPANDED REGISTER FILE

The standard register file of the Z8® has been expanded to form 16 Expanded Register File (ERF) Banks (Figure 2-5). Each ERF Bank consists of up to 256 registers (the same amount as in the Standard Register File) that can then be divided into 16 Working Register Groups. This expansion allows for access to additional feature/peripheral control and data registers.


REGISTER POINTER
7
6
5
4
3
2
1
0
Working Register Group Pointer
Expanded Register Group Pointer
Z8 Register File
FF
FO
7F
0F
00

EXPANDED REGISTER FILE BANK (F)

(F) 0F	WDTMR
(F) 0E	Reserved
(F) 0D	Reserved
(F) 0C	Reserved
(F) 0B	SMR
(F) 0A	Reserved
(F) 09	Reserved
(F) 08	Reserved
(F) 07	Reserved
(F) 06	Reserved
(F) 05	Reserved
(F) 04	Reserved
(F) 03	Reserved
(F) 02	Reserved
(F) 01	Reserved
(F) 00	PCON

EXPANDED REGISTER FILE BANK (C)

(C) 0F	Reserved
(C) 0C	Reserved
(C) 0D	Reserved
(C) 0C	Reserved
(C) 0B	Reserved
(C) 0A	Reserved
(C) 09	Reserved
(C) 08	Reserved
(C) 07	Reserved
(C) 06	Reserved
(C) 05	Reserved
(C) 04	Reserved
(C) 03	Reserved
(C) 02	SCON
(C) 01	RXBUF
(C) 00	SCOMP

EXPANDED REGISTER FILE BANK (0)

(0) 0F	GPR
(0) 0E	GPR
(0) 0D	GPR
(0) 0C	GPR
(0) 0B	GPR
(0) 0A	GPR
(0) 09	GPR
(0) 08	GPR
(0) 07	GPR
(0) 06	GPR
(0) 05	GPR
(0) 04	GPR
(0) 03	P3
(0) 02	P2
(0) 01	P1
(0) 00	P0

Figure 2-5. Expanded Register File Architecture

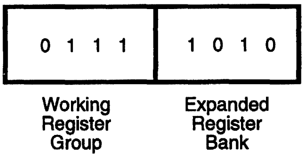
Note: The fully implemented register file is shown. Please refer to the specific product specification for actual register file architecture implemented.

Currently, three out of the possible sixteen Z8® ERF Banks have been implemented. ERF Bank 0, also known as the Z8 Standard Register File, has all 256 bytes defined (Figure 2-1). Only Working Register Group 0 (register addresses 00H to 0FH) have been defined for ERF Bank C and ERF Bank F (Table 2-4). All other working register groups in ERF Banks C and F, as well as the remaining thirteen ERF Banks, are not implemented. All are reserved for future use.

When an ERF Bank is selected, register addresses 00H to 0FH access those sixteen ERF Bank registers - in effect replacing the first sixteen locations of the Z8 Standard Register File.

For example, if ERF Bank C is selected, the Z8 Standard Registers 00H through 0FH are no longer accessible. Registers 00H through 0FH are now the 16 registers from ERF Bank C, Working Register Group 0. No other Z8 Standard Registers are effected since only Working Register Group 0 is implemented in ERF Bank C.

Access to the ERF is accomplished through the Register Pointer (FDH). The lower nibble of the Register Pointer determines the ERF Bank while the upper nibble determines the Working Register Group within the register file (Figure 2-6).



Selects ERF Bank A(H),
Working Register Group 7(H)

Figure 2-6. Register Pointer (FDH) Example

The value of the lower nibble in the Register Pointer (FDH) corresponds to the ERF Bank identification. Table 2.3 shows the lower nibble value and the register file assigned to it.

Table 2-3. ERF Bank Address

Register Pointer (FDH) Low Nibble	Hex	Register File
0000(B)	0	Z8® Standard Register File *
0001(B)	1	Expanded Register File Bank 1
0010(B)	2	Expanded Register File Bank 2
0011(B)	3	Expanded Register File Bank 3
0100(B)	4	Expanded Register File Bank 4
0101(B)	5	Expanded Register File Bank 5
0110(B)	6	Expanded Register File Bank 6
0111(B)	7	Expanded Register File Bank 7
1000(B)	8	Expanded Register File Bank 8
1001(B)	9	Expanded Register File Bank 9
1010(B)	A	Expanded Register File Bank A
1011(B)	B	Expanded Register File Bank B
1100(B)	C	Expanded Register File Bank C
1101(B)	D	Expanded Register File Bank D
1110(B)	E	Expanded Register File Bank E
1111(B)	F	Expanded Register File Bank F

Note: The Z8 Standard Register File is equivalent to Expanded Register File Bank 0.

The upper nibble of the register pointer selects which group of 16 bytes in the Register File, out of the full 256, will be accessed as working registers.

For example:
(See Figure 2-4)

```
R253 RP = 00H    ;ERF Bank 0, Working Reg. Group 0.
                 R0 = Port 0 = 00H
                 R1 = Port 1 = 01H
                 R2 = Port 2 = 02H
                 R3 = Port 3 = 03H
                 R11 = GPR 0BH
                 R15 = GPR 0FH
If:
R253 RP = 0FH    ;ERF Bank F, Working Reg. Group 0.
                 R0 = PCON = 00H
                 R1 = Reserved = 01H
                 R2 = Reserved = 02H
                 R11 = SMR = 0BH
                 R15 = WDTMR = 0FH

If:
R253 RP = FFH    ;ERF Bank F, Working Reg. Group F.
                 00H = PCON
                 R0 = SIO    01H= Reserved
                 R1 = TMR    02H= Reserved
                             ...
                 R2 = T1     0BH = SMR
                             ...
                 R15 = SPL   0FH = WDTMR
```

Note that since enabling an ERF Bank (C or F) only changes register addresses 00H to 0FH, the working register pointer can be used to access either the selected ERF Bank (Bank C or F, Working Register Group 0) or the Z8 Standard Register File (ERF Bank 0, Working Register Groups 1 through F).

Note: When an ERF Bank other than Bank 0 is enabled, the first 16 bytes of the Z8 Standard Register File (I/O ports 0 to 3, Groups 4 to F) are no longer accessible (the selected ERF Bank, Registers 00H to 0FH are accessed instead). It is important to re-initialize the Register Pointer to enable ERF Bank 0 when these registers are required for use.

The SPI register is mapped into ERF Bank C. Access is easily done using the following example:

```
LD    RP, #0CH     ;Select ERF Bank C working
                   ;register group 0 for access.
LD    R2,#xx       ;access SCON
LD    R1, #xx      ;access RXBUF
LD    RP, #00H     ;Select ERF Bank 0 so I/O ports
                   ;are again accessible.
```

Table 2-4. Z8 Expanded Register File Bank Layout

Expanded Register File Bank	ERF
F(H)	PCON, SMR, WDT, (00H, 0BH, 0FH), Working Register Group 0 only implemented.
E(H)	Not Implemented (Reserved)
D(H)	Not Implemented (Reserved)
C(H)	SPI Registers: SCOMP, RXBUF, SCON (00H, 01H, 02H), Working Register Group 0 only implemented.
B(H)	Not Implemented (Reserved)
A(H)	Not Implemented (Reserved)
9(H)	Not Implemented (Reserved)
8(H)	Not Implemented (Reserved)
7(H)	Not Implemented (Reserved)
6(H)	Not Implemented (Reserved)
5(H)	Not Implemented (Reserved)
4(H)	Not Implemented (Reserved)
3(H)	Not Implemented (Reserved)
2(H)	Not Implemented (Reserved)
1(H)	Not Implemented (Reserved)
0(H)	Z8 Ports 0, 1, 2, 3, and General-Purpose Registers 04H to EFH, and control registers F0H to FFH.

Please refer to the specific product specification to determine the above registers are implemented.

2.4 Z8 CONTROL AND PERIPHERAL REGISTERS

2.4.1 Standard Z8 Registers

The standard Z8® control registers govern the operation of the CPU. Any instruction which references the register file can access these control registers. Available control registers are:

- Interrupt Priority Register (IPR)
- Interrupt Mask Register (IMR)
- Interrupt Request Register (IRQ)
- Program Control Flags (FLAGS)
- Register Pointer (RP)
- Stack Pointer High-Byte (SPH)
- Stack Pointer Low-Byte (SPL)

The Z8 uses a 16-bit Program Counter (PC) to determine the sequence of current program instructions. The PC is not an addressable register.

Peripheral registers are used to transfer data, configure the operating mode, and control the operation of the on-chip peripherals. Any instruction that references the register file can access the peripheral registers. The peripheral registers are:

- Serial I/O (SIO)
- Timer Mode (TMR)
- Timer/Counter 0 (T0)
- T0 Prescaler (PRE0)
- Timer/Counter 1 (T1)
- T1 Prescaler (PRE1)
- Port 0-1 Mode (P01M)
- Port 2 Mode (P2M)
- Port 3 Mode (P3M)

In addition, the four port registers (P0-P3) are considered to be peripheral registers.

2.4.2 Expanded Z8 Registers

The expanded Z8 control registers govern the operation of additional features or peripherals. Any instruction which references the register file can access these registers.

The ERF contains the control registers for WDT, Port Control, Serial Peripheral Interface (SPI), and the SMR functions. Figure 2-4 shows the layout of the Register Banks in the ERF. Register Bank C in the ERF consists of the registers for the SPI. Table 2-5 shows the registers within ERF Bank C, Working Register Group 0.

Table 2-5. Expanded Register File Register Bank C, WR Group 0

Register	Register Function	Working Register
F	Reserved	R15
E	Reserved	R14
D	Reserved	R13
C	Reserved	R12
B	Reserved	R11
A	Reserved	R10
9	Reserved	R9
8	Reserved	R8
7	Reserved	R7
6	Reserved	R6
5	Reserved	R5
4	Reserved	R4
3	Reserved	R3
2	SPI Control (SCON)	R2
1	SPI Tx/Rx Data (RxBuf)	R1
0	SPI Compare (SCOMP)	R0

Working Register Group 0 in ERF Bank 0 consists of the registers for Z8 General-Purpose Registers and ports. Table 2-6 shows the registers within this group.

Table 2-6. Expanded Register File Bank 0, WR Group 0

Register	Register Function	Working Register
F	General-Purpose Register	R15
E	General-Purpose Register	R14
D	General-Purpose Register	R13
C	General-Purpose Register	R12
B	General-Purpose Register	R11
A	General-Purpose Register	R10
9	General-Purpose Register	R9
8	General-Purpose Register	R8
7	General-Purpose Register	R7
6	General-Purpose Register	R6
5	General-Purpose Register	R5
4	General-Purpose Register	R4
3	Port 3	R3
2	Port 2	R2
1	Port 1	R1
0	Port 0	R0

Working Register Group 0 in ERF Bank F consists of the control registers for STOP mode, WDT, and port control. Table 2-7 shows the registers within this group.

Table 2-7. Expanded Register File Bank F, WR Group 0

Register	Register Function	Working Register
F	WDTMR	R15
E	Reserved	R14
D	Reserved	R13
C	Reserved	R12
B	SMR	R11
A	Reserved	R10
9	Reserved	R9
8	Reserved	R8
7	Reserved	R7
6	Reserved	R6
5	Reserved	R5
4	Reserved	R4
3	Reserved	R3
2	Reserved	R2
1	Reserved	R1
0	PCON	R0

The functions and applications of the control and peripheral registers are described in subsequent sections of this manual.

2.5 PROGRAM MEMORY

The first 12 bytes of Program Memory are reserved for the interrupt vectors (Figure 2-7). These locations contain six 16-bit vectors that correspond to the six available interrupts. Address 12 up to the maximum ROM address consists of on-chip mask-programmable ROM. See the product data sheet for the exact program, data, register memory size, and address range available. At addresses outside the internal ROM, the Z8® executes external program memory fetches through Port 0 and Port 1 in Address/Data mode for devices with Port 0 and Port 1 featured. Otherwise, the program counter will continue to execute NOPs up to address FFFFH, roll over to 0000H, and continue to fetch executable code (Figure 2-7).

The internal program memory is one-time programmable (OTP) or mask programmable dependent on the specific device. ***A ROM protect feature prevents "dumping" of the ROM contents by inhibiting execution of the LDC, LDCI, LDE, and LDEI instructions to Program Memory in all modes. ROM look-up tables cannot be used with this feature.***

The ROM Protect option is mask-programmable, to be selected by the customer when the ROM code is submitted. For the OTP ROM, the ROM Protect option is an OTP programming option.

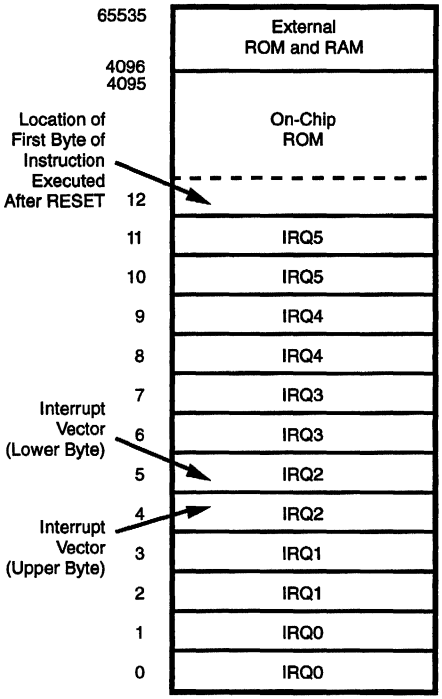

65535
External ROM and RAM
4096
4095
On-Chip ROM
Location of First Byte of Instruction Executed After RESET
12
11
IRQ5
10
IRQ5
9
IRQ4
8
IRQ4
7
IRQ3
Interrupt Vector (Lower Byte)
6
IRQ3
5
IRQ2
4
IRQ2
Interrupt Vector (Upper Byte)
3
IRQ1
2
IRQ1
1
IRQ0
0
IRQ0

Figure 2-7. Z8 Program Memory Map

2.6 Z8 EXTERNAL MEMORY

The Z8®, in some cases, has the capability to access external program memory with the 16-bit Program Counter. To access external program memory the Z8 offers multiplexed address/data lines (AD7-AD0) on Port 1 and address lines (A15-A8) on Port 0. This feature only applies to devices that offer Port 0 and Port 1. The maximum external address is FFFF. This memory interface is supported by the control lines /AS (Address Strobe), /DS (Data Strobe), and R/W (Read/Write). The origin of the external program memory starts after the last address of the internal ROM. Figure 2-8 shows an example of external program memory for the Z8.

2.6.1 External Data Memory (/DM)

The Z8, in some cases, can address up to 60 Kbytes of external data memory beginning at location 4096. External Data Memory may be included with, or separated from, the external Program Memory space. /DM, an optional I/O function that can be programmed to appear on pin P34, is used to distinguish between data and program memory space. The state of the /DM signal is controlled by the type of instruction being executed. An LDC opcode references Program (/DM inactive) Memory, and an LDE instruction references Data (/DM active Low) Memory. The user must configure Port 3 Mode Register (P3M) bits D3 and D4 for this mode.

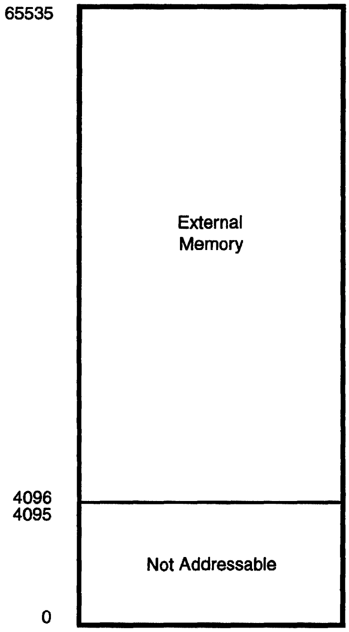


Figure 2-8. External Memory Map

Note: For additional information on using external memory, see Chapter 10 of this manual. For exact memory addressing options available, see the device product specification.

2.7 Z8 STACKS

Stack operations can occur in either the Z8® Standard Register File or external data memory. Under software control, Port 0-1 Mode register (F8H) selects the stack location. Only the General-Purpose Registers can be used for the stack when the internal stack is selected.

The register pair FEH and FFH form the 16-bit Stack Pointer (SP), that is used for all stack operations. The stack address is stored with the MSB in FEH and LSB in FFH (Figure 2-9).

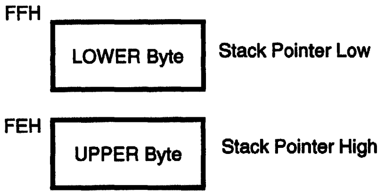


Figure 2-9. Stack Pointer

The stack address is decremented prior to a PUSH operation and incremented after a POP operation. The stack address always points to the data stored on the top of the stack. The Z8® stack is a return stack for CALL instructions and interrupts, as well as a data stack.

During a CALL instruction, the contents of the PC are saved on the stack. The PC is restored during a RETURN instruction. Interrupts cause the contents of the PC and Flag registers to be saved on the stack. The IRET instruction restores them (Figure 2-10).

When the Z8 is configured for an internal stack (using the Z8 Standard Register File), register FFH serves as the Stack Pointer. The value in FEH is ignored. FEH can be used as a general-purpose register in this case only.

An overflow or underflow can occur when the stack address is incremented or decremented during normal stack operations. The programmer must prevent this occurrence or unpredictable operation will result.

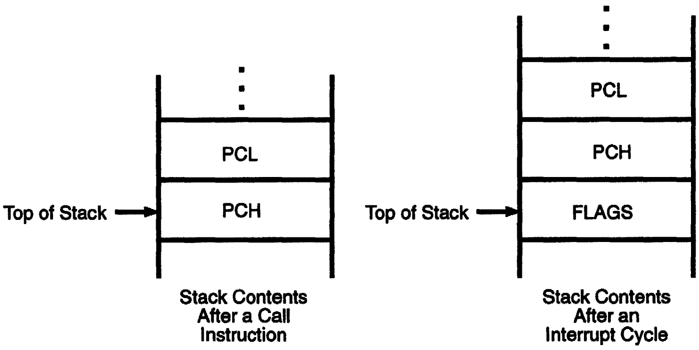

PCL
PCH
Top of Stack
Stack Contents After a Call Instruction
PCL
PCH
FLAGS
Top of Stack
Stack Contents After an Interrupt Cycle

Figure 2-10. Stack Operations

CHAPTER 3
CLOCK

3.1 CLOCK

The Z8® derives its timing from on-board clock circuitry connected to pins XTAL1 and XTAL2. The clock circuitry consists of an oscillator, a divide-by-two shaping circuit, and a clock buffer. Figure 3-1 illustrates the clock circuitry. The oscillator's input is XTAL1 and its output is XTAL2. The clock can be driven by a crystal, a ceramic resonator, LC clock, RC, or an external clock source.

3.1.1 Frequency Control

In some cases, the Z8 has an EPROM/OTP option or a Mask ROM option bit to bypass the divide-by-two flip flop in Figure 3-1. This feature is used in conjunction with the low EMI option. When low EMI is selected, the device output drive and oscillator drive is reduced to approximately 25 percent of the standard drive and the divide-by-two flip flop is bypassed such that the XTAL clock frequency is equal to the internal system clock frequency. In this mode, the maximum frequency of the XTAL clock is 4 MHz. Please refer to specific product specification for availability of options and output drive characteristics.

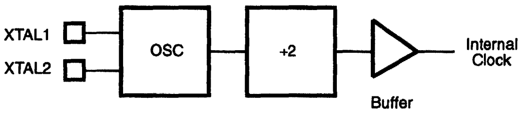
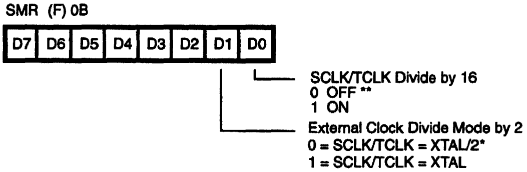


Figure 3-1. Z8 Clock Circuit

3.2 CLOCK CONTROL

In some cases, the Z8 offers software control of the internal system clock via programming register bits. The bits are located in the Stop-Mode Recovery Register in Expanded Register File Bank F, Register 0BH. This register selects the clock divide value and determines the mode of Stop-Mode Recovery (Figure 3-2). Please refer to the specific product specification for availability of this feature/register.



* Default setting after RESET.
** Default setting after RESET and STOP-Mode Recovery.

Figure 3-2. Stop-Mode Recovery Register (Write-Only Except D7, Which is Read-Only)

3.2.1 SCLK/TCLK Divide-By-16 Select (D0)

This bit of the SMR controls a divide-by-16 prescalar of SCLK/TCLK. The purpose of this control is to selectively reduce device power consumption during normal processor execution (SCLK control) and/or HALT mode (where TCLK sources counter/timers and interrupt logic).

3.2.2 External Clock Divide-By-Two (D1)

This bit can eliminate the oscillator divide-by-two circuitry. When this bit is 0, SCLK (System Clock) and TCLK (Timer Clock) are equal to the external clock frequency divided by two. The SCLK/TCLK is equal to the external clock frequency when this bit is set (D1 = 1). Using this bit, together with D7 of PCON, further helps lower EMI (D7 (PCON) = 0, D1 (SMR) = 1). The default setting is 0. Maximum frequency is 4 MHz with D1 = 1 (Figure 3-3).

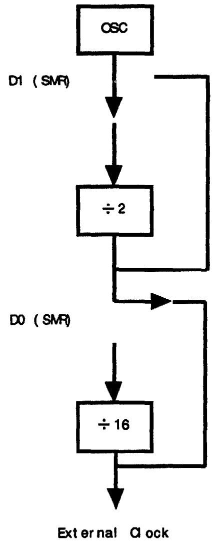


Figure 3-3. External Clock Circuit

3.3 Oscillator Control

In some cases, the Z8® offers software control of the oscillator to select low EMI drive or standard drive. The selection is done by programming bit D7 of the Port Configuration (PCON) register (Figure 3-4). The PCON register is located in Expanded Register File Bank F, Register 00H.

A 1 in bit D7 configures the oscillator with standard drive, while a 0 configures the oscillator with Low EMI drive. This only affects the drive capability of the oscillator and does not affect the relationship of the XTAL clock frequency to the internal system clock (SCLK).

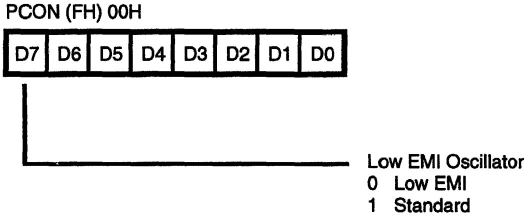


Figure 3-4. Port configuration register (PCON (Write-Only)

3.4 OSCILLATOR OPERATION

The Z8® uses a Pierce oscillator with an internal feedback (Figure 3-5). The advantages of this circuit are low cost, large output signal, low-power level in the crystal, stability with respect to V_{CC} and temperature, and low impedances (not disturbed by stray effects).

One draw back is the need for high gain in the amplifier to compensate for feedback path losses. The oscillator amplifies its own noise at start-up until it settles at the frequency that satisfies the gain/phase requirements A x B = 1, where A = V_0/V_i is the gain of the amplifier and B = V_i/V_0 is the gain of the feedback element. The total phase shift around the loop is forced to zero (360 degrees). Since V_{IN} must be in phase with itself, the amplifier/inverter provides 180 degree phase shift and the feedback element is forced to provide the other 180 degrees of phase shift.

R_1 is a resistive component placed from output to input of the amplifier. The purpose of this feedback is to bias the amplifier in its linear region and to provide the start-up transition.

Capacitor C_2 combined with the amplifier output resistance provides a small phase shift. It will also provide some attenuation of overtones.

Capacitor C_1 combined with the crystal resistance provides additional phase shift.

C_1 and C_2 can affect the start-up time if they increase dramatically in size. As C_1 and C_2 increase, the start-up time increases until the oscillator reaches a point where it does not start up any more.

It is recommended for fast and reliable oscillator start-up (over the manufacturing process range) that the load capacitors be sized as low as possible without resulting in overtone operation.

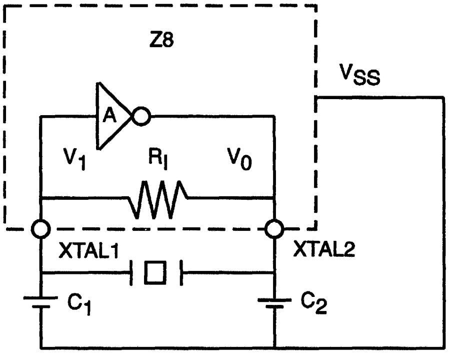


Figure 3-5. Pierce Oscillator with Internal Feedback Circuit

3.4.1 Layout

Traces connecting crystal, caps, and the Z8® oscillator pins should be as short and wide as possible. This reduces parasitic inductance and resistance. The components (caps, crystal, resistors) should be placed as close as possible to the oscillator pins of the Z8.

The traces from the oscillator pins of the IC and the ground side of the lead caps should be guarded from all other traces (clock, V_{CC}, address/data lines, system ground) to reduce cross talk and noise injection. This is usually accomplished by keeping other traces and system ground trace planes away from the oscillator circuit and by placing a Z8 device V_{SS} ground ring around the traces/components. The ground side of the oscillator lead caps should be connected to a single trace to the Z8 V_{SS} (GND) pin. It should not be shared with any other system ground trace or components except at the Z8 device V_{SS} pin. This is to prevent differential system ground noise injection into the oscillator (Figure 3-6).

3.4.2 Indications of an Unreliable Design

There are two major indicators that are used in working designs to determine their reliability over full lot and temperature variations. They are:

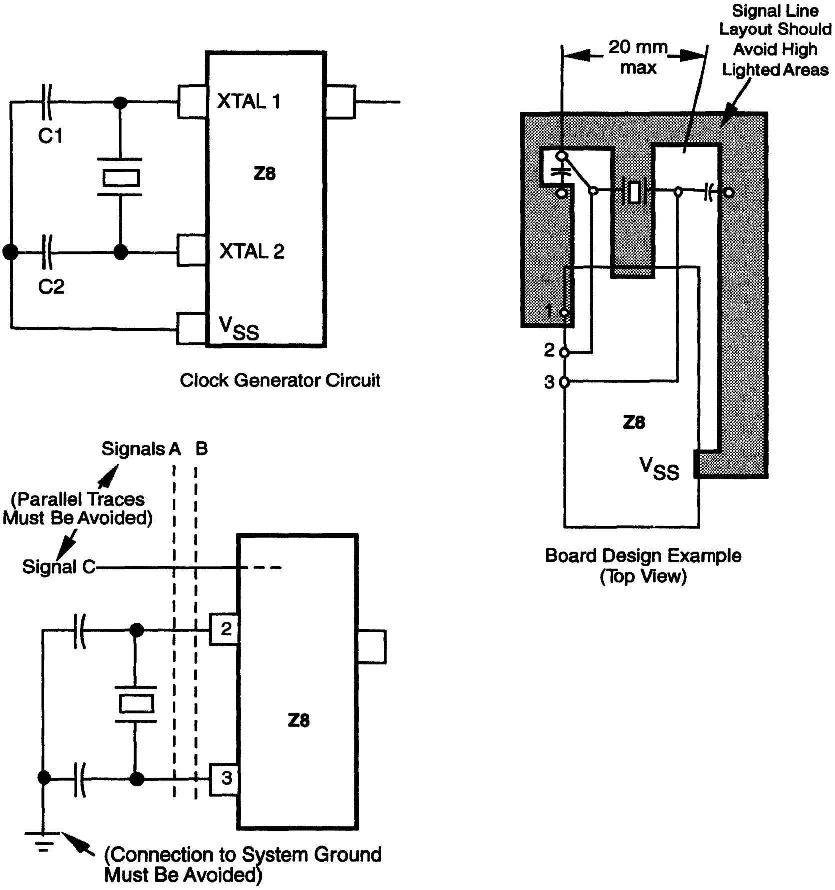
Start-up Time. If start -up time is excessive, or varies widely from unit to unit, there is probably a gain problem. C_1/C_2 needs to be reduced; the amplifier gain is not adequate at frequency, or crystal Rs is too large.

Output Level. The signal at the amplifier output should swing from ground to V_{CC}. This indicates there is adequate gain in the amplifier. As the oscillator starts up, the signal amplitude grows until clipping occurs, at which point the loop gain is effectively reduced to unity and constant oscillation is achieved. A signal of less than 2.5 volts peak-to-peak is an indication that low gain may be a problem. Either C_1 or C_2 should be made smaller or a low-resistance crystal should be used.

3.4.3 Circuit Board Design Rules

The following circuit board design rules are suggested:

- To prevent induced noise the crystal and load capacitors should be physically located as close to the Z8® as possible.
- Signal lines should not run parallel to the clock oscillator inputs. In particular, the crystal input circuitry and the internal system clock output should be separated as much as possible.
- V_{CC} power lines should be separated from the clock oscillator input circuitry.
- Resistivity between XTAL1 or XTAL2 and the other pins should be greater than 10 Mohms.


C1
XTAL 1
Z8
XTAL 2
C2
VSS
Signal Line Layout Should Avoid High Lighted Areas
20 mm max
1
2
3
Z8
VSS
Signals A B
(Parallel Traces Must Be Avoided)
Signal C
2
3
Z8
(Connection to System Ground Must Be Avoided)

Clock Generator Circuit

Board Design Example (Top View)

Figure 3-6. Circuit Board Design Rules

3.4.4 Crystals and Resonators

Crystals and ceramic resonators (Figure 3-7) should have the following characteristics to ensure proper oscillator operation:

Crystal Cut	AT (crystal only)
Mode	Parallel, Fundamental Mode
CrystalCapacitance	<7pF
Load Capacitance	10pF < CL < 220 pF, 15 typical
Resistance	100 ohms max

Depending on operation frequency, the oscillator may require the addition of capacitors C_1 and C_2 (shown in Figures 3-7). The capacitance values are dependent on the manufacturer's crystal specifications.

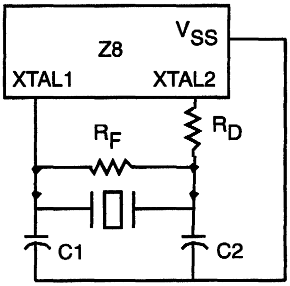


Figure 3-7. Crystal/Ceramic Resonator Oscillator

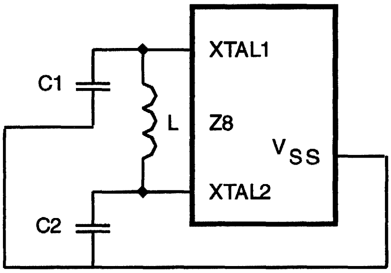


Figure 3-8. LC Clock

In most cases, the R_D is 0 Ohms and R_F is infinite. It is determined and specified by the crystal/ceramic resonator manufacturer. The R_D can be increased to decrease the amount of drive from the oscillator output to the crystal. It can also be used as an adjustment to avoid clipping of the oscillator signal to reduce noise. The R_F can be used to improve the start-up of the crystal/ceramic resonator. The Z8 oscillator already has an internal shunt resistor in parallel to the crystal/ceramic resonator.

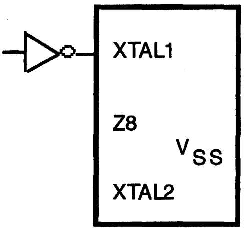


Figure 3-9. External Clock

It is recommended in Figures 3-7, 3-8, and 3-9 to connect the load capacitor ground trace directly to the V_{SS} (GND) pin of the Z8®. This ensures that no system noise is injected into the Z8 clock. This trace should not be shared with any other components except at the V_{SS} pin of the Z8.

In some cases, the Z8 XTAL1 pin also functions as one of the EPROM high-voltage mode programming pins or as a special factory test pin. In this case, applying 2 V above V_{CC} on the XTAL1 pin will cause the device to enter one of these modes. Since this pin accepts high voltages to enter these respective modes, the standard input protection diode to V_{CC} is not on XTAL1. It is recommended that in applications where the Z8 is exposed to much system noise, a diode from XTAL1 to V_{CC} be used to prevent accidental enabling of these modes. This diode will not affect the crystal/ceramic resonator operation .

Please note that a parallel resonant crystal or resonator data sheet will specify a load capacitor value that is the series combination of C_1 and C_2, including all parasitics (PCB and holder).

3.5 LC OSCILLATOR.

The Z8 oscillator can use a LC network to generate a XTAL clock (Figure 3-8).

The frequency stays stable over V_{CC} and temperature. The oscillation frequency is determined by the equation:

$$\text{Frequency} = \frac{1}{2\pi(\text{LCT})1/2}$$

where L is the total inductance including parasitics and C_T is the total series capacitance including the parasitics.

Simple series capacitance is calculated using the following equation:

$$\frac{1}{C_T} = \frac{1}{C_1} + \frac{1}{C_2}$$

If $C_1 = C_2$

$$\frac{1}{C_T} = \frac{2}{C_1}$$

$$C_1 = 2CT$$

Sample calculation of capacitance C_1 and C_2 for 5.83 MHz frequency and inductance value of 27 uH:

$$5.83\,(10\text{^}6) = \frac{1}{2\pi\,[2.7\,(10^{-6})\,C_T]\ 1/2}$$

$$CT = 27.6\text{ pf}$$

Thus C_1 = 55.2 pf and C_2 = 55.2 pf.

3.6 RC OSCILLATOR

In some cases, the Z8® has a RC oscillator option. Please refer to the specific product specification for availability. The RC oscillator requires a resistor across XTAL1 and XTAL2. An additional load capacitor is required from the XTAL1 input to V_{SS} pin (Figure 3-9).

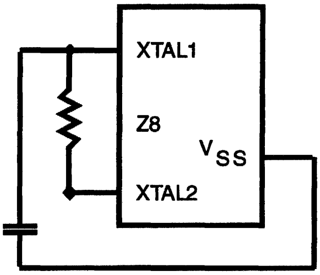


Figure 3-9. RC Clock

CHAPTER 4
RESET—WATCH-DOG TIMER

4.1 RESET

This section describes the Z8® reset conditions, reset timing, and register initialization procedures. Reset is generated by Power-On Reset (POR), Reset Pin, Watch-Dog Timer (WDT), and Stop-Mode Recovery.

A system reset overrides all other operating conditions and puts the Z8 into a known state. To initialize the chip's internal logic, the /RESET input must be held Low for at least 4 internal system clock periods. The control register and ports are reset to their default conditions after a POR, a reset from the /Reset pin, or Watch-Dog Timer timeout while in RUN mode and HALT mode. The control registers and ports are not reset to their default conditions after Stop- Mode Recovery and WDT timeout while in STOP mode.

While /RESET is Low, /AS is output at the internal clock rate, /DS is forced Low, and R//W remains High. The program counter is loaded with 000CH. I/O ports and control registers are configured to their default reset state.

Resetting the Z8 does not effect the contents of the general-purpose registers.

4.2 /Reset Pin, Internal POR Operation

In some cases, the Z8 hardware /RESET pin initializes the control and peripheral registers, as shown in Tables 4-1, 4-2, 4-3, and 4-4. Specific reset values are shown by 1 or 0, while bits whose states are unknown are indicated by the letter U. The Tables 4-1, 4-2, 4-3, and 4-4 show the reset conditions for the generic Z8.

Note: The register file reset state is device dependent. Please refer to the selected device product specifications for register availability and reset state.

Table 4-1. Sample Control and Peripheral Register Reset Values (ERF Bank 0)

Register (HEX)	Register Name	7	6	5	4	3	2	1	0	Comments
F0	Serial I/O	U	U	U	U	U	U	U	U	
F1	Timer Mode	0	0	0	0	0	0	0	0	Counter/Timers Stopped
F2	Counter/Timer1	U	U	U	U	U	U	U	U	
F3	T1 Prescaler	U	U	U	U	U	U	0	0	Single-Pass Count Mode, External Clock Source
F4	Counter/Timer0	U	U	U	U	U	U	U	U	
F5	T0 Prescaler	U	U	U	U	U	U	U	0	Single-Pass Count Mode
F6	Port 2 Mode	1	1	1	1	1	1	1	1	All Inputs
F7	Port 3 Mode	0	0	0	0	0	0	0	0	Port 2 Open-Drain, P33-P30 Input, P37-P34 Output
F8	Port 0-1 Mode	0	1	0	0	1	1	0	1	Internal Stack, Normal Memory Timing
F9	Interrupt Priority	U	U	U	U	U	U	U	U	
FA	Interrupt Request	0	0	0	0	0	0	0	0	All Interrupts Cleared
FB	Interrupt Mask	0	U	U	U	U	U	U	U	Interrupts Disabled
FC	Flags	U	U	U	U	U	U	U	U	
FD	Register Pointer	0	0	0	0	0	0	0	0	
FE	Stack Pointer (High)	U	U	U	U	U	U	U	U	
FF	Stack Pointer (Low)	U	U	U	U	U	U	U	U	

Program execution starts 5 to 10 clock cycles after /RESET has returned High. The initial instruction fetch is from location 000CH. Figure 4-1 shows reset timing.

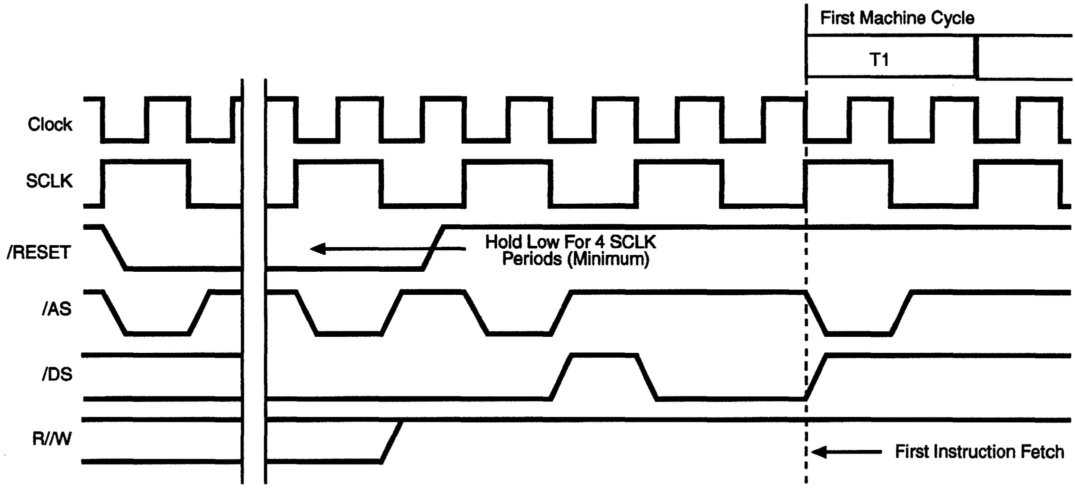


Figure 4-1. Reset Timing

After a reset, the first routine executed should be one that initializes the control registers to the required system configuration.

The /RESET pin is the input of a Schmitt-triggered circuit. Resetting the Z8® will initialize port and control registers to their default states. To form the internal reset line, the output of the trigger is synchronized with the internal clock. The clock must therefore be running for /RESET to function. It requires 4 internal system clocks after reset is detected for the Z8 to reset the internal circuitry. An internal pull-up, combined with an external capacitor of 1 uf, provides enough time to properly reset the Z8 (Figure 4-2). In some cases, the Z8 has an internal POR timer circuit that holds the Z8 in reset mode for a duration (T_{POR}) before releasing the device out of reset. On these Z8 devices, the internally generated reset drives the reset pin low for the POR time. Any devices driving the reset line must be open-drained in order to avoid damage from possible conflict during reset conditions. This reset time allows the on-board clock oscillator to stabilize.

To avoid asynchronous and noisy reset problems, the Z8 is equipped with a reset filter of four external clocks (4TpC). If the external reset signal is less than 4TpC in duration, no reset occurs. On the fifth clock after the reset is detected, an internal RST signal is latched and held for an internal register count of 18 external clocks, or for the duration of the external reset, whichever is longer. During the reset cycle, /DS is held active low while /AS cycles at a rate of the internal system clock. Program execution begins at location 000CH, 5-10 TpC cycles after /RESET is released. For the internal Power-On Reset, the reset output time is specified as T_{POR}. Please refer to specific product specifications for actual values.

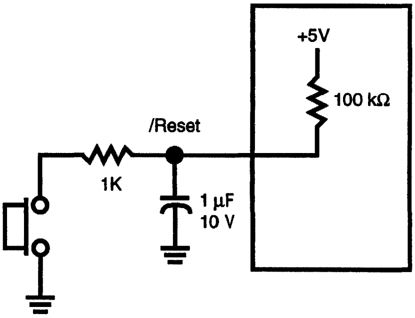


Figure 4-2. Example of External Power-On Reset Circuit

Table 4-2. Sample Expanded Register File Bank 0 Reset Values

Register (HEX)	Register Name	Bits 7	6	5	4	3	2	1	0	Comments
00	Port 0	U	U	U	U	U	U	U	U	Input mode, output set to push-pull
01	Port 1	U	U	U	U	U	U	U	U	Input mode, output set to push-pull
02	Port 2	U	U	U	U	U	U	U	U	Input mode, output set to push-pull
03	Port 3	1	1	1	1	U	U	U	U	Standard Digital input and output
04-EF	General-Purpose Registers 04-EF	U	U	U	U	U	U	U	U	Standard Digital input and output

Table 4-3. Sample Expanded Register File Bank C Reset Values

Register (HEX)	Register Name	Bits 7	6	5	4	3	2	1	0	Comments
00	SPI Compare (SCOMP)	0	0	0	0	0	0	0	0	
01	Receive Buffer (RxBUF)	U	U	U	U	U	U	U	U	
02	SPI Control (SCON)	U	U	U	U	0	0	0	0	

Table 4-4. Sample Expanded Register File Bank F Reset Values

Register (HEX)	Register Name	Bits 7	6	5	4	3	2	1	0	Comments
00	Port Configuration (PCON)	1	1	1	1	1	1	1	0	Comparator outputs disabled on Port 3 Port 0 and 1 output is push-pull Port 0, 1, 2, 3, and oscillator with standard output drive
0B	STOP-Mode Recovery (SMR)	0	0	1	0	0	0	0	0	Clock divide by 16 off XTAL divide by 2 POR and / OR External Reset Stop delay on Stop recovery level is low, STOP flag is POR
0F	Watch-Dog Timer Mode (WDTMR)	U	U	U	0	1	1	0	1	512 T_pC for WDT time out, WDT runs during STOP and HALT mode, on-board RC drives WDT

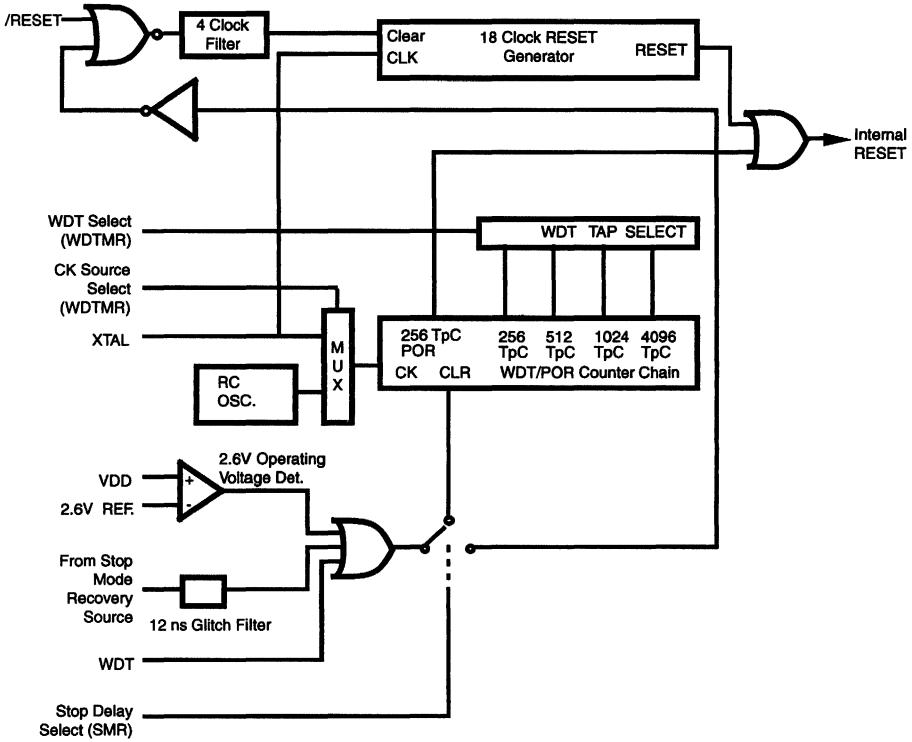


Figure 4-3. Example of Z8 Reset with /RESET Pin, WDT, SMR, and POR

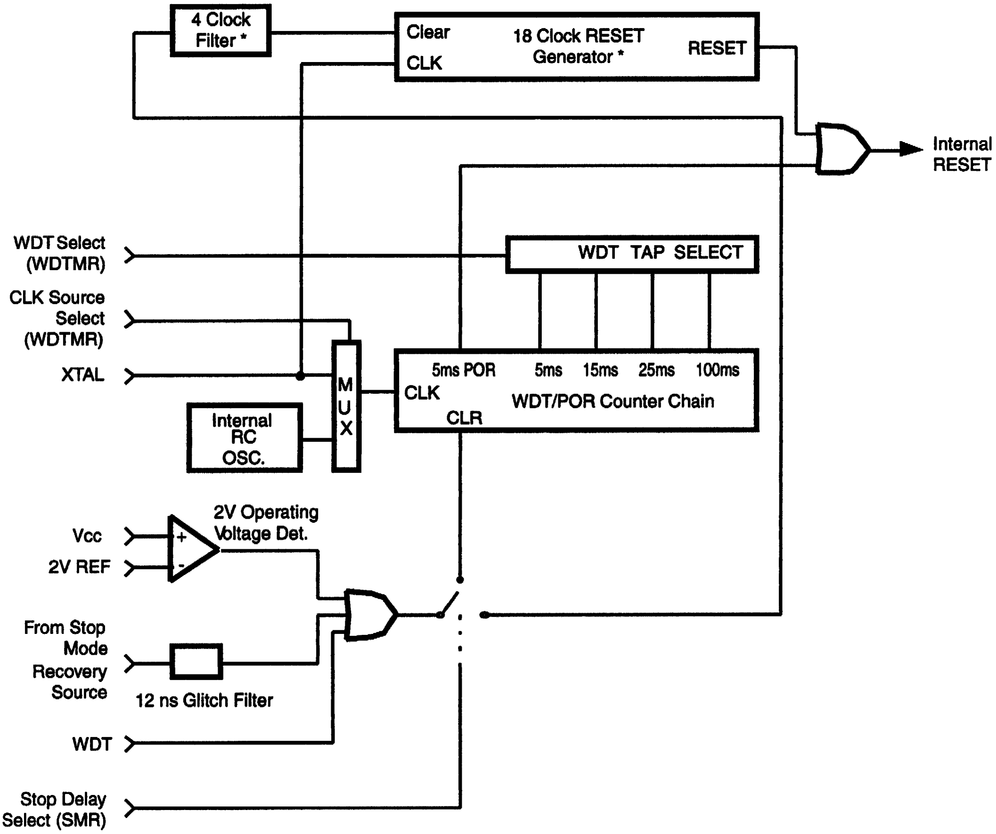
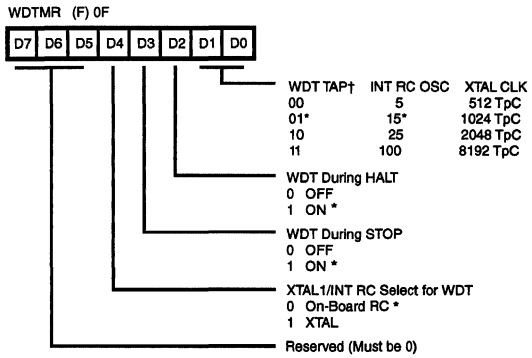


Figure 4-4. Example of Z8 Reset with WDT, SMR, and POR

4.3 Watch-Dog Timer (WDT)

The WDT is a retriggerable one-shot timer that resets the Z8® if it reaches its terminal count. When operating in the RUN or HALT modes, a WDT reset is functionally equivalent to a hardware /POR reset. The WDT is initially enabled by executing the WDT instruction and refreshed on subsequent executions of the WDT instruction. The WDT cannot be disabled after it has been initially enabled. Permanently enabled WDTs are always enabled and the WDT instruction is used to refresh it. The WDT circuit is driven by an onboard RC oscillator or external oscillator from the XTAL1 pin. The POR clock source is selected with bit 4 of the Watch-Dog Timer Mode register (WDTMR). In some cases, a Z8 that offers the WDT but does not have a WDTMR register, has a fixed WDT timeout and uses the on board RC oscillator as the only clock source. Please refer to specific product specifications for selectability of timeout, WDT during HALT and STOP modes, source of WDT clock, and availability of the permanently-on WDT option.

Note: Execution of the WDT instruction affects the Z (zero), S (sign), and V (overflow) flags.



* Default setting after RESET
† Must be 01 for Z86C03

Figure 4-5. Example of Z8 Watch-Dog Timer Mode Register (Write-Only)

Note: The WDTMR register is accessible only during the first 64 processor cycles from the execution of the first instruction after Power-On Reset, Watch-Dog Reset or a Stop-Mode Recovery. After this point, the register cannot be modified by any means, intentional or otherwise. The WDTMR is a write-only register.

The WDTMR is located in Expanded Register File Bank F, register 0FH. The control bits are described as follows:

WDT Time Select (D1, D0). Bits 0 and 1 control a tap circuit that determines the time-out period. Table 4-5 shows the different values that can be obtained. The default value of D1 and D0 are 0 and 1, respectively.

Table 4-5. Time-Out Period of the WDT

Time-Out of		Typical Time-Out of	
D1	**D0**	**Internal RC OSC**	**XTAL Clock**
0	0	5 ms min	256TpC
0	1	15 ms min	512TpC
1	0	25 ms min	1024TpC
1	1	100 ms min	4096TpC

Notes:
TpC = XTAL clock cycle
The default on reset is, D0 = 1 and D1 = 0.
The values given are for V_{CC} = 5.0V.
See the device product specification for exact WDTMR time-out select options available.

WDT During HALT (D2). This bit determines whether or not the WDT is active during HALT mode. A 1 indicates active during HALT. The default is 1. A WDT time out during HALT mode will reset control register ports to their default reset conditions.

WDT During STOP (D3). This bit determines whether or not the WDT is active during STOP mode. Since XTAL clock is stopped during STOP Mode, unless as specified below, the on-board RC must be selected as the clock source to the POR counter. A 1 indicates active during STOP. The default is 1. If bits D3 and D4 are both set to 1, the WDT only, is driven by the external clock during STOP mode. This feature makes it possible to wake up from STOP mode from an internal source. Please refer to specific product specifications for conditions of control and port registers when the Z8 comes out of STOP mode. A WDT time out during STOP mode will not reset all control registers. The reset conditions of the ports from STOP mode due to WDT time out is the same as if recovered using any of the other STOP mode sources.

Clock Source for WDT (D4). This bit determines which oscillator source is used to clock the internal POR and WDT counter chain. If the bit is a 1, the internal RC oscillator is bypassed and the POR and WDT clock source is driven from the external pin, XTAL1. The default configuration of this bit is 0, which selects the internal RC oscillator.

Bits 5, 6 and 7. These bits are reserved.

V_{CC} Voltage Comparator. An on-board voltage comparator checks that V_{CC} is at the required level to insure correct operation of the device. Reset is globally driven if V_{CC} is below the specified voltage. This feature is available in select ROM Z8® devices. See the device product specification for feature availability and operating range.

4.4 POWER-ON-RESET (POR)

A timer circuit clocked by a dedicated on-board RC oscillator is used for the Power-On Reset (POR) timer (T_{POR}) function. The POR time allows V_{CC} and the oscillator circuit to stabilize before instruction execution begins.

The POR timer circuit is a one-shot timer triggered by one of three conditions:

1. Power fail to Power OK status (cold start).
2. STOP-Mode Recovery (if bit 5 of SMR=1).
3. WDT timeout.

The POR time is specified as T_{POR}. On Z8 devices that feature a Stop-Mode Recovery register (SMR), bit 5 selects whether the POR timer is used after Stop-Mode Recovery or by-passed. If bit D5 = 1 then the POR timer is used. If bit 5 = 0 then the POR timer is by-passed. In this case, the Stop-Mode Recovery source must be held in the recovery state for 5 T_PC or 5 crystal clocks to pass the reset signal internally. This option is used when the clock is provided with an RC/LC clock. See the device product specification for timing details.

POR (cold start) will always reset the Z8 control and port registers to their default condition. If a Z8 has a SMR register, the warm start bit will be reset to a 0 to indicate POR.

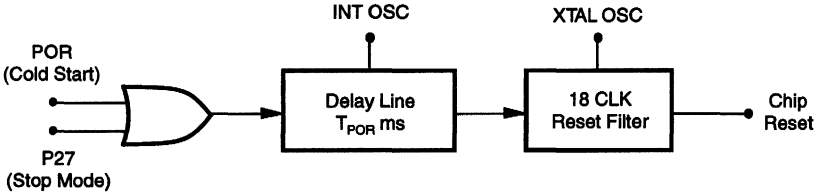


Figure 4-6. Example of Z8 with Simple SMR and POR

CHAPTER 5

I/O PORTS

5.1 INTRODUCTION

The Z8® has up to 32 lines dedicated to input and output. These lines are grouped into four 8-bit ports known as Port 0, Port 1, Port 2, and Port 3. Port 0 is nibble programmable as input, output, or address. Port 1 is byte configurable as input, output, or address/data. Port 2 is bit programmable as either inputs or outputs, with or without handshake and SPI. Port 3 can be programmed to provide timing, serial and parallel input/output, or comparator input/output.

All ports have push-pull CMOS outputs. In addition, the push-pull outputs of Port 2 can be turned off for open-drain operation.

5.1.1 Mode Registers

Each port has an associated Mode Register that determines the port's functions and allows dynamic change in port functions during program execution. Port and Mode Registers are mapped into the Standard Register File as shown in Figure 5-1.

Register	HEX	Identifier
Port 0-1 Mode	F8H	P01M
Port 3 Mode	F7H	P3M
Port 2 Mode	F6H	P2M
Port 3	03H	P3
Port 2	02H	P2
Port 1	01H	P1
Port 0	00H	P0

Figure 5-1. I/O Ports and Mode Registers

Because of their close association, Port and Mode Registers are treated like any other general-purpose register. There are no special instructions for port manipulation. Any instruction which addresses a register can address the ports. Data can be directly accessed in the Port Register, with no extra moves.

5.1.2 Input and Output Registers

Each bit of Ports 0, 1, and 2, have an input register, an output register, associated buffer, and control logic. Since there are separate input and output registers associated with each port, writing to bits defined as inputs stores the data in the output register. This data cannot be read as long as the bits are defined as inputs. However, if the bits are reconfigured as outputs, the data stored in the output register is reflected on the output pins and can then be read. This mechanism allows the user to initialize the outputs prior to driving their loads (Figure 5-2).

Since port inputs are asynchronous to the Z8 internal clock, a READ operation could occur during an input transition. In this case, the logic level might be uncertain (somewhere between a logic 1 and 0). To eliminate this meta-stable condition, the Z8 latches the input data two clock periods prior to the execution of the current instruction. The input register uses these two clock periods to stabilize to a legitimate logic level before the instruction reads the data.

Note: The following sections describe the generic function of the Z8 ports. Any additional features of the ports such as SPI, C/T, and Stop-Mode Recovery are covered in their own section.

5.2 Port 0

This section deals with only the I/O operation of Port 0. The port's external memory interface operation is covered later in this manual. Figure 5-2 shows a block diagram of Port 0. This diagram also applies to Ports 1 and 2.

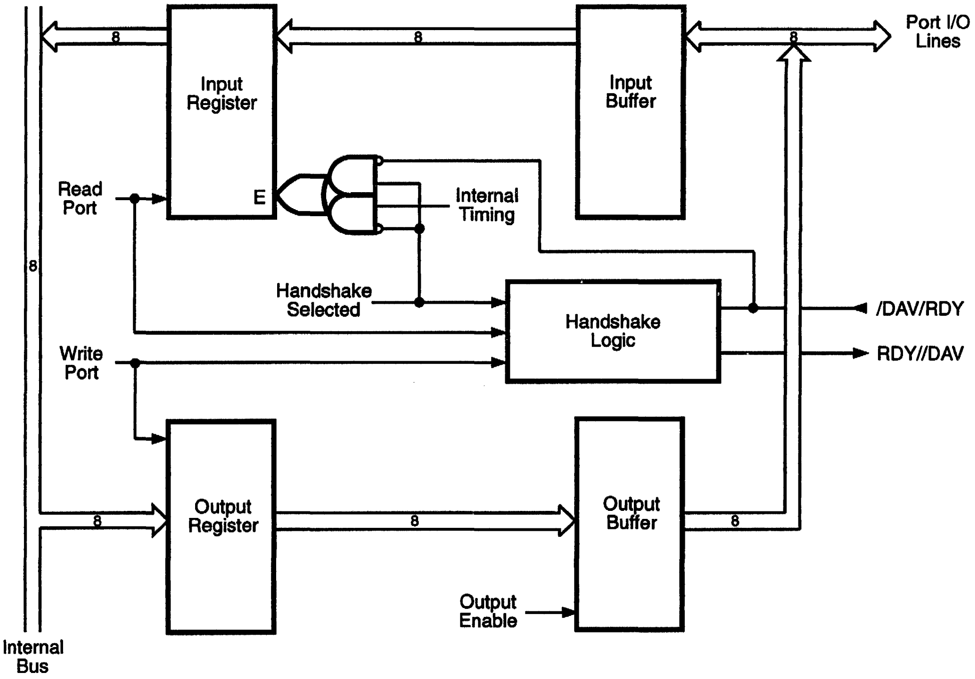
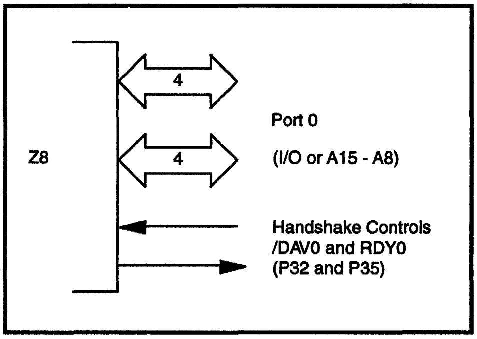


Figure 5-2. Ports 0, 1, 2 Generic Block Diagram

5.2.1 General I/O Mode

Port 0 can be an 8-bit, bidirectional, CMOS or TTL compatible I/O port. These eight I/O lines can be configured under software control as a nibble I/O port (P03-P00 input/output and P07-P04 input/output), or as an address port for interfacing external memory. The input buffers can be Schmitt-triggered, level shifted, or a single-trip point buffer and can be nibble programmed. Either nibble output can be globally programmed as push-pull or open-drain. Low EMI output buffers in some cases can be globally programmed by the software, as an OTP program option, or as a ROM mask option. In some, the Z8® has Auto Latches hardwired to the inputs. Please refer to specific product specifications for exact input/output buffer type features that are available (Figures 5-3a and 5-3b).



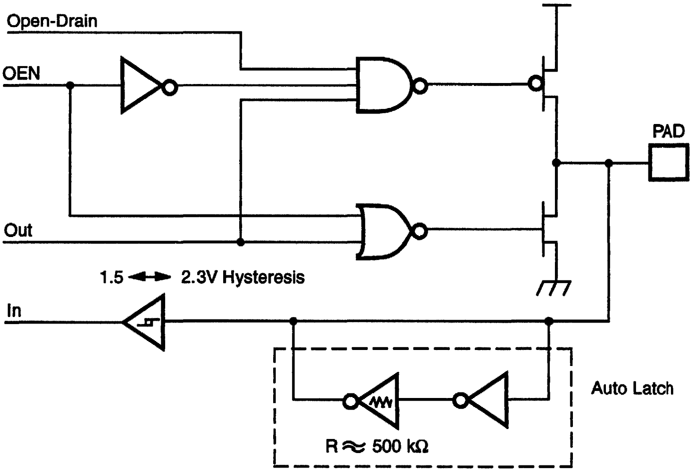


Figure 5-3a. Port 0 Configuration with Open-Drain Capability, Auto Latch, and Schmitt-Trigger

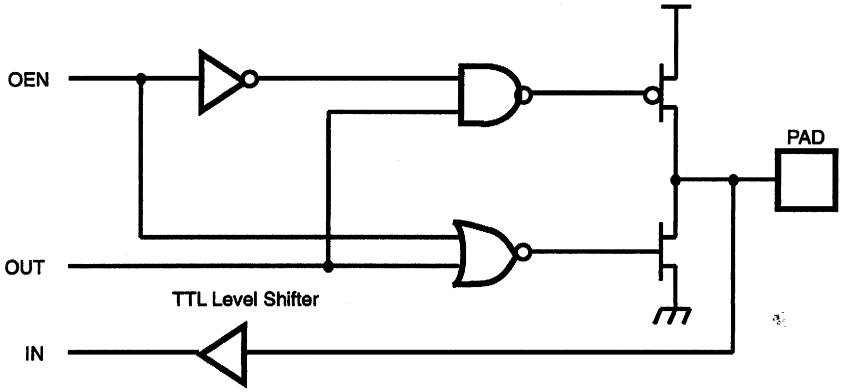


Figure 5-3b. Port 0 Configuration with TTL Level Shifter

5.2.2 Read/Write Operations

In the nibble I/O Mode, Port 0 is accessed as general-purpose register P0 (00H) with ERF Bank set to 0. The port is written by specifying P0 as an instruction's destination register. Writing to the port causes data to be stored in the port's output register.

The port is read by specifying P0 as the source register of an instruction. When an output nibble is read, data on the external pins is returned. Under normal loading conditions this is equivalent to reading the output register. However, for Port 0 outputs defined as open-drain, the data returned is the value forced on the output by the external system. This may not be the same as the data in the output register. Reading a nibble defined as input also returns data on the external pins. However, input bits under handshake control return data latched into the input register via the input strobe.

The Port 0-1 Mode resister bits D_1D_0 and D_7D_6 are used to configure Port 0 nibbles. The lower nibble ($P0_0$-$P0_3$) can be defined as inputs by setting bits D_1 to 0 and D_0 to 1, or as outputs by setting both D_1 and D_0 to 0. Likewise, the upper nibble ($P0_4$-$P0_7$) can be defined as inputs by setting bits D_7 to 0 and D_6 to 1, or as outputs by setting both D_6 and D_7 to 0 (Figure 5-4).

5.2.3 Handshake Operation

When used as an I/O port, Port 0 can be placed under handshake control by programming the Port 3 Mode register bit D_2 to 1. In this configuration, handshake control lines are DAV_0 ($P3_2$) and RDY_0 ($P3_5$) when Port 0 is an input port, or RDY_0 ($P3_2$) and DAV_0 ($P3_5$) when Port 0 is an output port. (See Figure 5-5.)

Handshake direction is determined by the configuration (input or output) assigned to the Port 0 upper nibble, $P0_4$-$P0_7$. The lower nibble must have the same I/O configuration as the upper nibble to be under handshake control. Figure 5-3a illustrates the Port 0 upper and lower nibbles and the associated handshake lines of Port 3.

5.3 Port 1

This section deals only with the I/O operation. The port's external memory interface operation is discussed later in this manual. Figure 5-2 shows a block diagram of Port 1.

5.3.1 General I/O Mode

Port 1 can be an 8-bit, bidirectional, CMOS or TTL compatible port with multiplexed Address (A7-A0) and Data (D7-D0) ports. These eight I/O lines can be byte programmed as inputs or outputs or can be configured under software control as an Address/Data port for interfacing to external memory. The input buffers can be Schmitt-triggered, level-shifted, or a single-point buffer. In some cases, the output buffers can be globally programmed as either push-pull or open-drain. Low-EMI output buffers can be globally programmed by software, as an OTP program option, or as a ROM Mask Option. In some cases, the Z8® can have auto latches hardwired to the inputs. Please refer to specific product specifications for exact input/output buffer-type features available (Figures 5-6a and 5-6b).

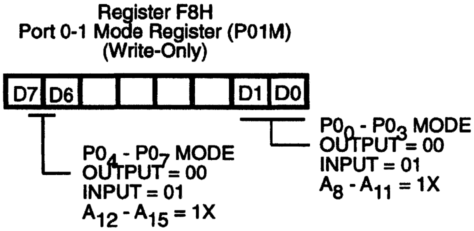


Figure 5-4. Port 0 I/O Operation

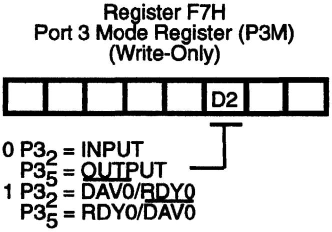


Figure 5-5. Port 0 Handshake Operation

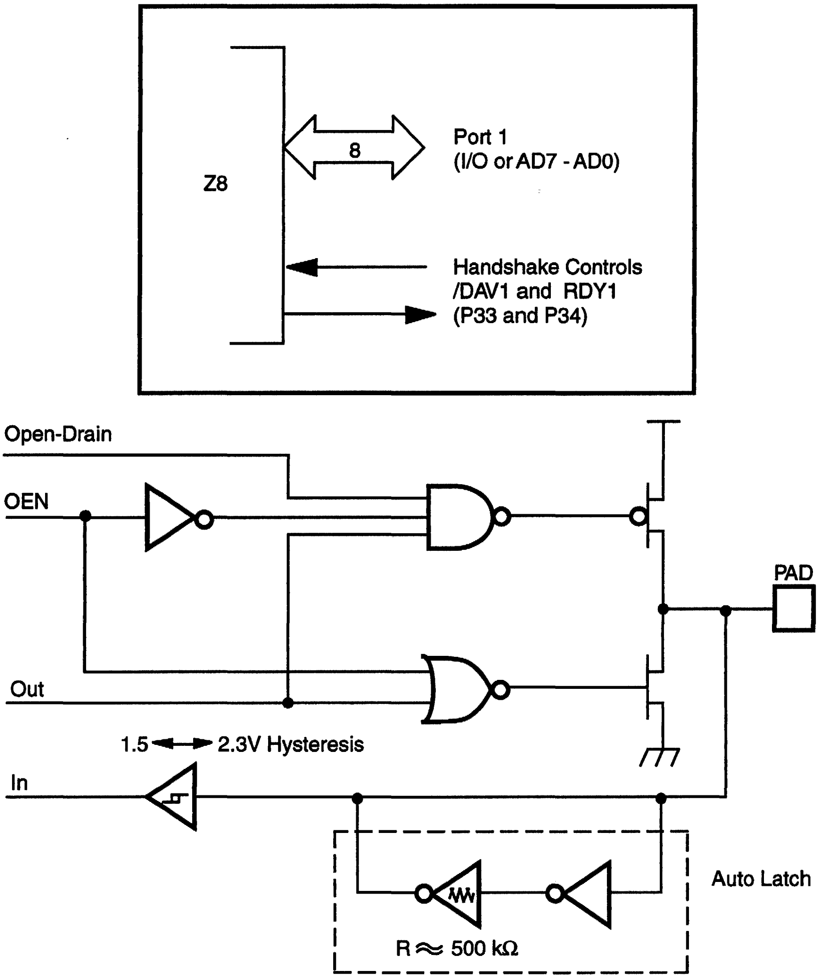


Figure 5-6a. Port 1 Configuration with Open-Drain Capability, Auto Latch, and Schmitt-Trigger

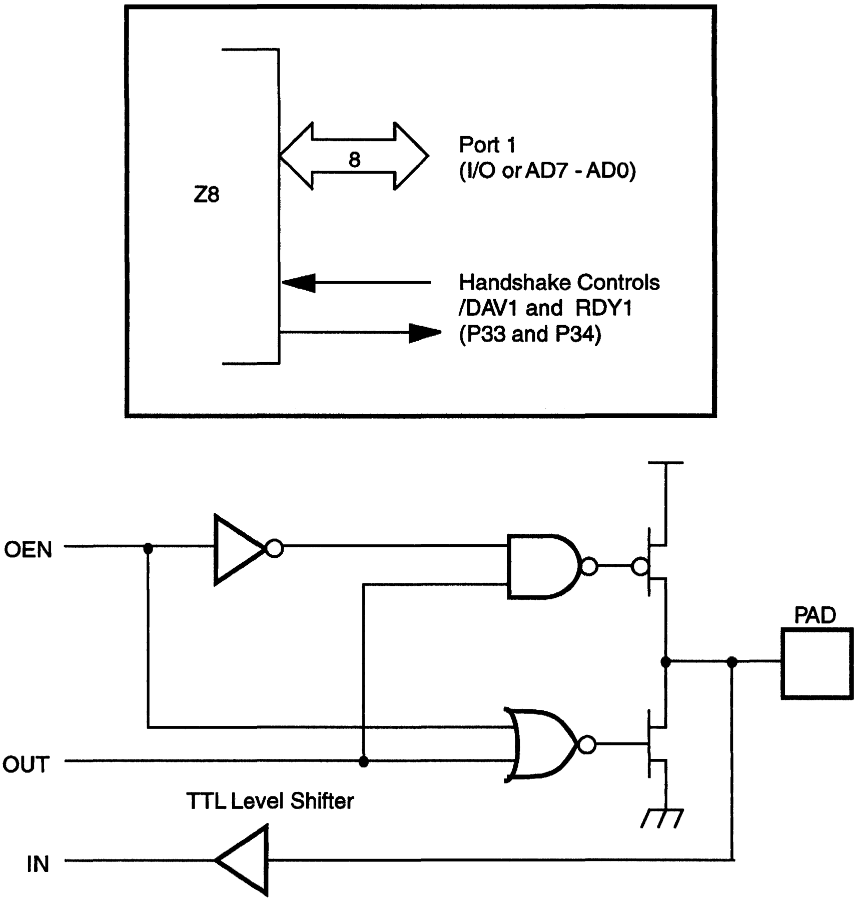


Figure 5-6b. Port 1 Configuration with TTL Level Shifter

5.3.2 Read/Write Operations

In byte input or byte output mode, the port is accessed as General-Purpose Register P1 (01H). The port is written by specifying P1 as an instruction's destination register. Writing to the port causes data to be stored in the port's output register.

The port is read by specifying P1 as the source register of an instruction. When an output is read, data on the external pins is returned. Under normal loading conditions, this is equivalent to reading the output register. However, if Port 1 outputs are defined as open-drain, the data returned is the value forced on the output by the external system. This may not be the same as the data in the output register. When Port 1 is defined as an input, reading also returns data on the external pins. However, inputs under handshake control return data latched into the input register via the input strobe.

Using the Port 0-1 Mode Register, Port 1 is configured as an output port by setting bits D_4 and D_3 to 0, or as an input port by setting D_4 to 0 and D_3 to 1 (Figure 5-8).

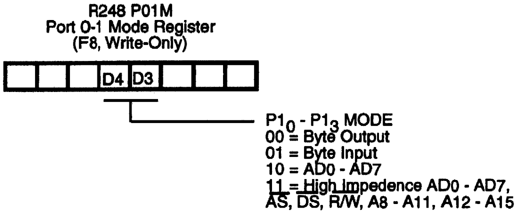


Figure 5-7. Port 1 I/O Operation

5.3.3 Handshake Operations

When used as an I/O port, Port 1 can be placed under handshake control by programming the Port 3 Mode register bits D_4 and D_3 both to 1. In this configuration, handshake control lines are DAV$_1$ (P3$_3$) and RDY$_1$ (P3$_4$) when Port 1 is an input port, or RDY$_1$ (P3$_3$) and DAV$_1$ (P3$_4$) when Port 1 is an output port. See Figures 5-6 and 5-8.

Handshake direction is determined by the configuration (input and output) assigned to Port 1. For example, if Port 1 is an output port then handshake is defined as output.

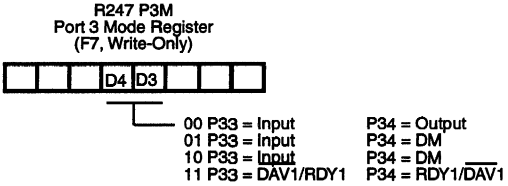


Figure 5-8. Handshake Operation

5.4 PORT 2

Port 2 is a general-purpose port. Figure 5-2 shows a block diagram of Port 2. Each of its lines can be independently programmed as input or output via the Port 2 Mode Register (F6H) as seen in Figure 5-9. A bit set to a 1 in P2M configures the corresponding bit in Port 2 as an input, while a bit set to 0 configures an output line.

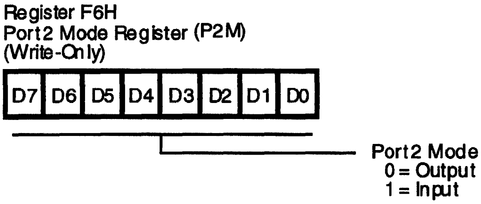


Figure 5-9. Port 2 I/O Mode Configuration

5.4.1 General Port I/O

Port 2 can be an 8-bit, bidirectional, CMOS- or TTL-compatible I/O port. These eight I/O lines can be configured under software control to be an input or output, independently. Input buffers can be Schmitt-triggered, level-shifted, or a single trip point buffer and may contain Auto Latches. Bits programmed as outputs may be globally programmed as either push-pull or open-drain. Low-EMI output buffers can be globally programmed by the software, an OTP program option, or as a ROM mask option. In addition, when the SPI is featured and enabled, P20 functions as data-in (DI), and P27 functions as data-out (DO). Please refer to specific product specifications for exact input/output buffer type features available. See Figures 5-10a through 5-10c.

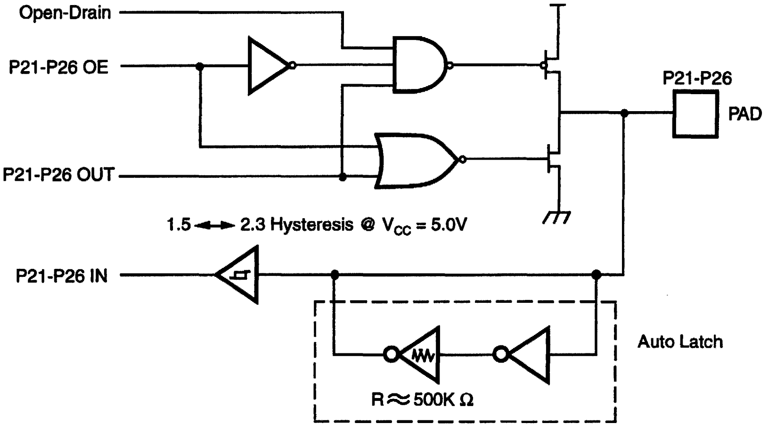


Figure 5-10a. Port 2 Configuration with Open-Drain Capability, Auto Latch, and Schmitt-Trigger

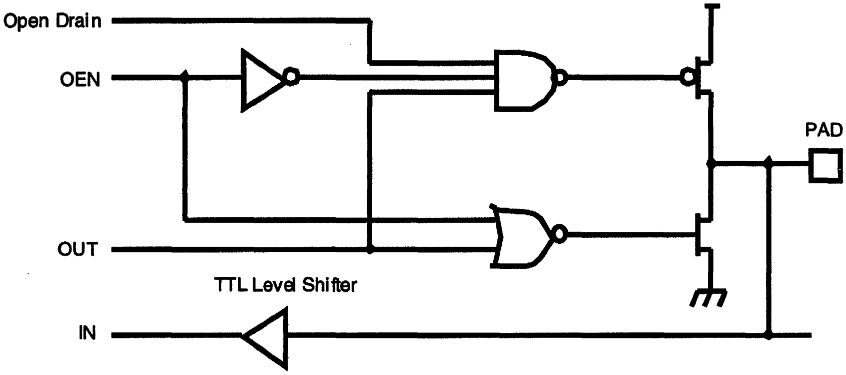


Figure 5-10b. Port 2 Configuration with TTL Level Shifter

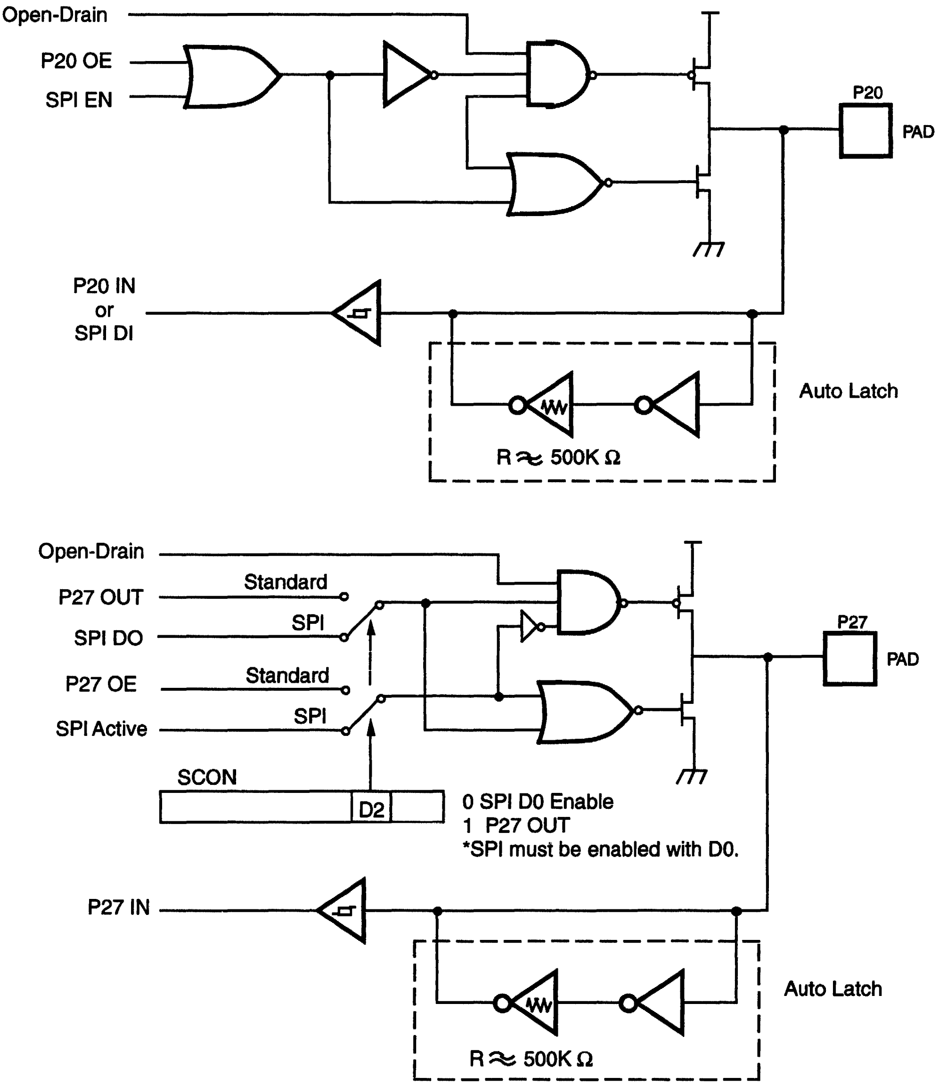


Figure 5-10c. Port 2 Configuration with Open-Drain Capability, Auto Latch, Schmitt-Trigger and SPI

5.4.2 Read/Write Operations

Port 2 is accessed as General-Purpose Register P2 (02H). Port 2 is written by specifying P2 as an instruction's destination register. Writing to Port 2 causes data to be stored in the output register of Port 2, and reflected externally on any bit configured as an output. Regardless of the bit input/output configuration, Port 2 is always written and read as a byte-wide port.

Port 2 is read by specifying P2 as the source register of an instruction. When an output bit is read, data on the external pin is returned. Under normal loading conditions, this is equivalent to reading the output register. However, if a bit of Port 2 is defined as an open-drain output, the data returned is the value forced on the output pin by the external system. This may not be the same as the data in the output register. Reading input bits of Port 2 also returns data on the external pins. However, inputs under handshake control return data latched into the input register via the input strobe.

5.4.3 Handshake Operation

Port 2 can be placed under handshake control by programming bit 6 in the Port 3 Mode Register (Figure 5-11). In this configuration, Port 3 lines P31 and P36 are used as the handshake control lines /DAV2 and RDY2 for input handshake, or RDY2 and /DAV2 for output handshake.

Handshake direction is determined by the configuration (input or output) assigned to bit 7 of Port 2. Only those bits with the same configuration as P27 will be under handshake control. Figure 5-12 illustrates bit lines of Port 2 and the associated handshake lines of Port 3.

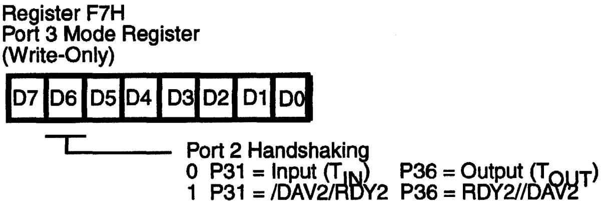


Figure 5-11. Port 2 Handshake Configuration

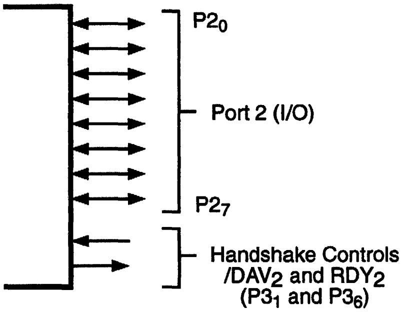


Figure 5-12. Port 2 Handshaking

5.5 PORT 3

5.5.1 General Port I/O

Port 3 differs structurally from Port 0, 1, and 2. Port 3 lines are fixed as four inputs (P33-P30) and four outputs (P37-P34) Port 3 does not have an input and output register for each bit. Instead, all the input lines have one input register, and all the output lines have an output register. Port 3 can be a CMOS- or TTL- compatible I/O port. Under software control, the lines can be configured as special control lines for handshake, comparator inputs, SPI control, external memory status, or I/O lines for the on-board serial and timer facilities. Figure 5-13 is a generic block diagram of Port 3. The inputs can be Schmitt-triggered, level-shifted, or single-trip point buffered. In some cases, the Z8® may have auto latches hardwired on certain Port 3 inputs and Low-EMI capabilities on the outputs. Please refer to specific product specifications for exact input/output buffer type features. Please refer to the section on counter/timers, Stop-Mode Recovery, serial I/O, comparators, and interrupts for more information on the relationships of Port 3 to that feature.

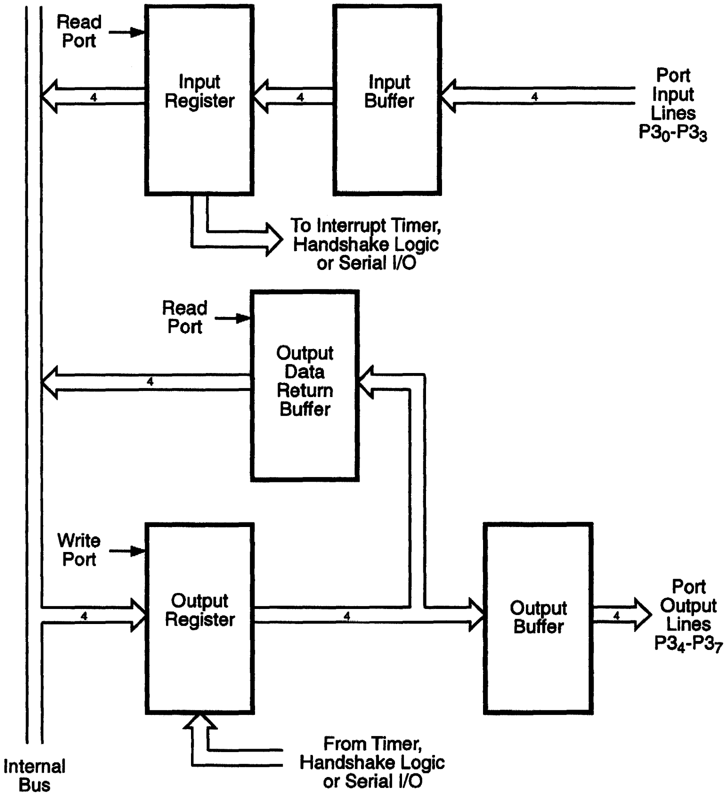

Read Port
Input Register
4
Input Buffer
4
4
Port Input Lines P3₀-P3₃
To Interrupt Timer, Handshake Logic or Serial I/O
Read Port
Output Data Return Buffer
4
Write Port
Output Register
4
4
Output Buffer
4
Port Output Lines P3₄-P3₇
From Timer, Handshake Logic or Serial I/O
Internal Bus

Figure 5-13. Port 3 Block Diagram

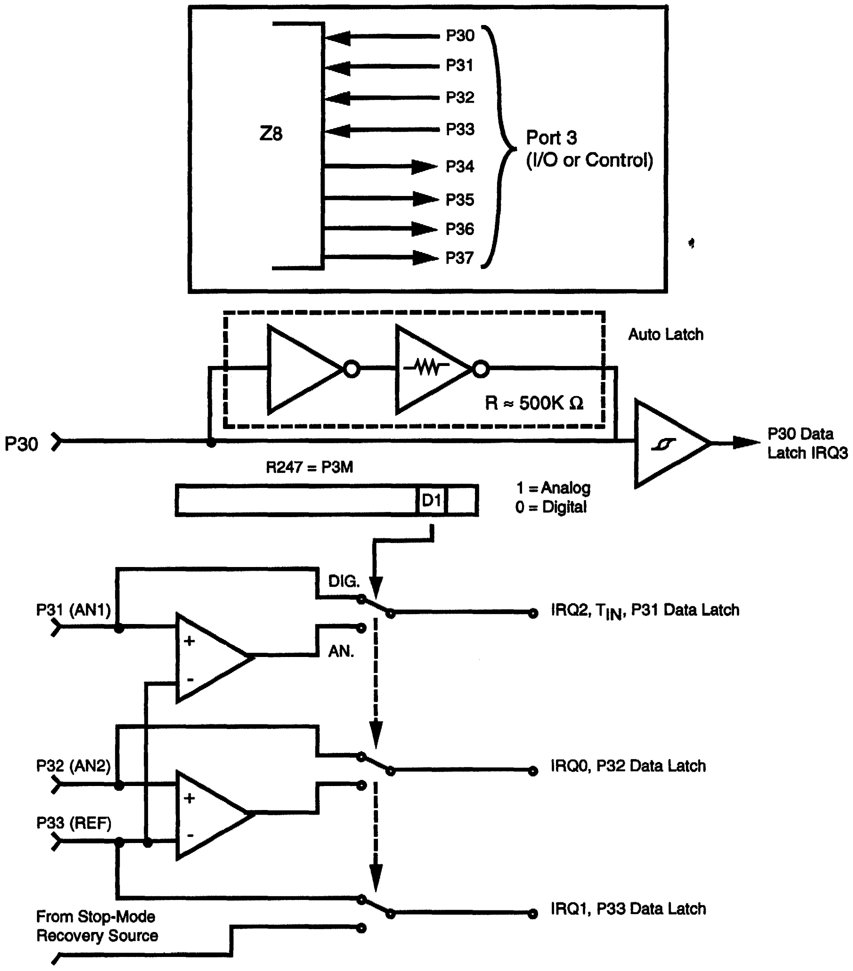


Figure 5-14a. Port 3 Configuration with Comparator, Auto Latch, and Schmitt-Trigger

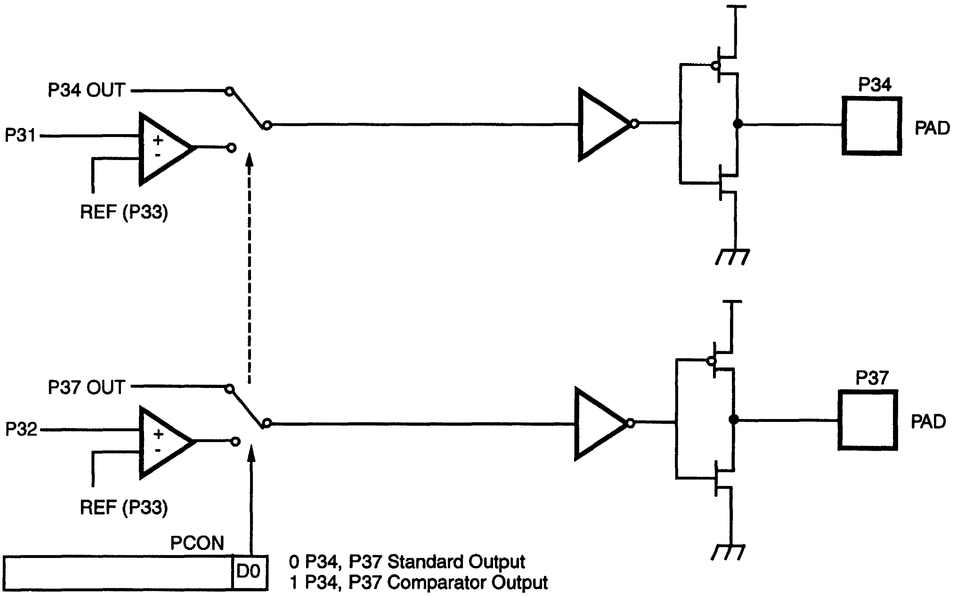


Figure 5-14b. Port 3 Configuration with Comparator

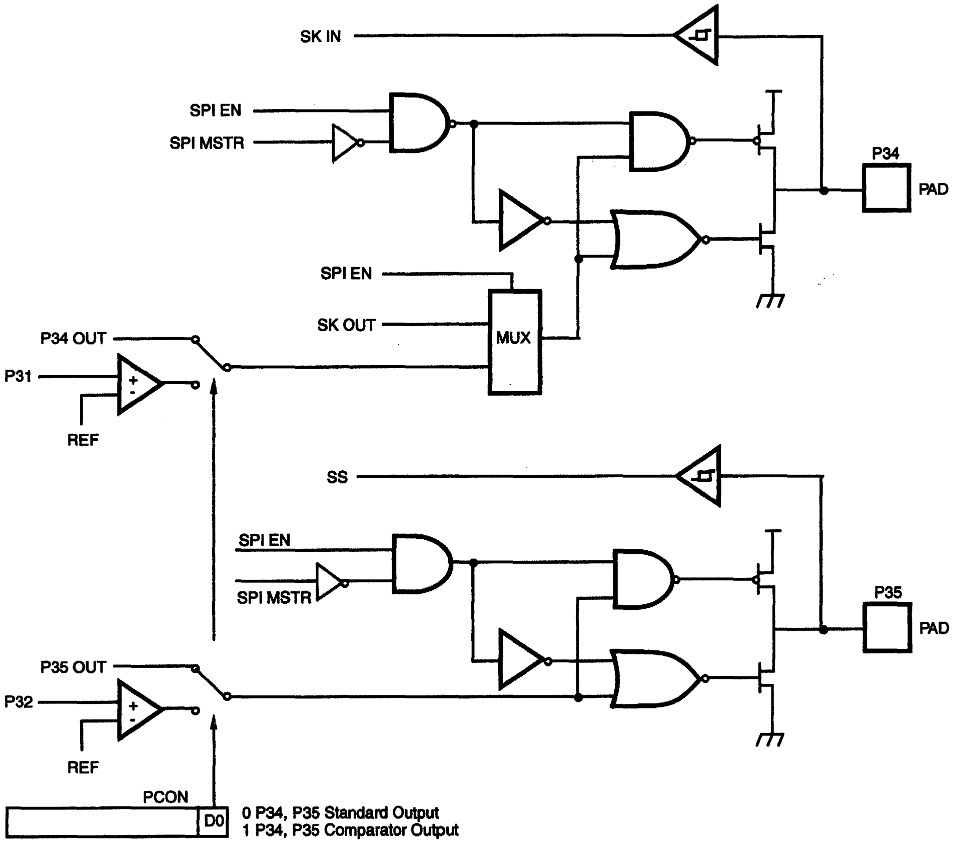


Figure 5-14c. Port 3 Configuration with SPI and Comparator Outputs Using P34 and P35

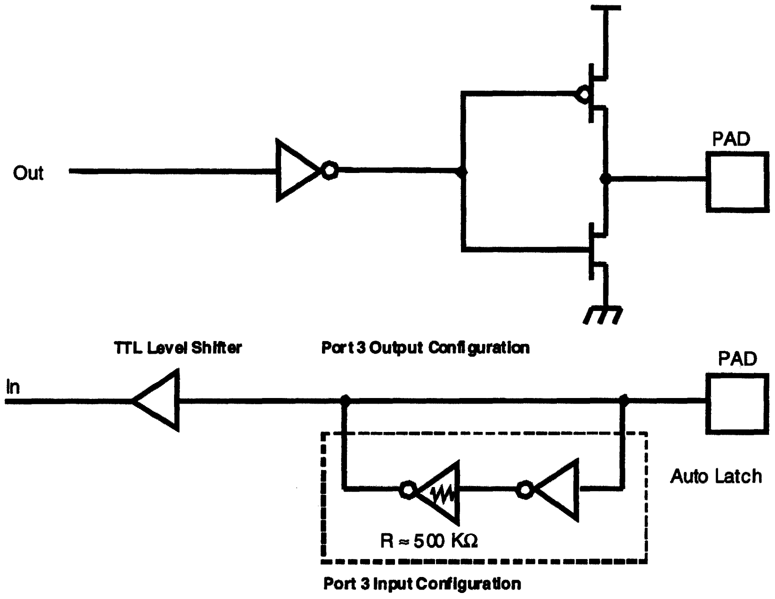


Figure 5-14d. Port 3 Configuration with TTL Level Shifter and Auto Latch

5.5.2 Read/Write Operations

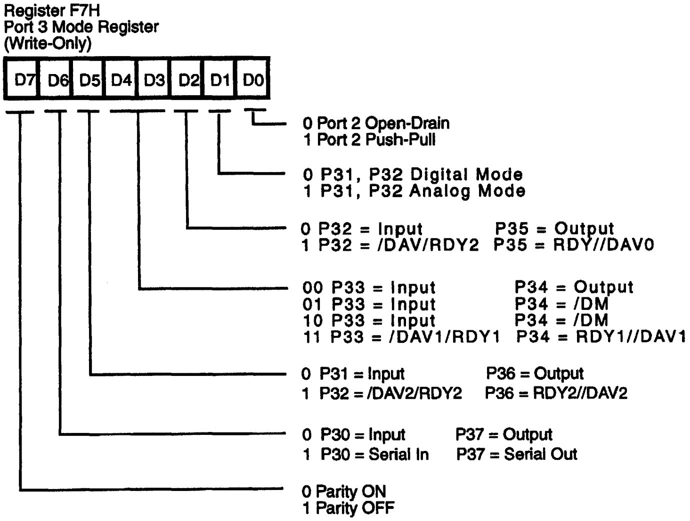
Port 3 is accessed as a General-Purpose Register P3 (03H). Port 3 is written by specifying P3 as an instruction's destination register. However, Port 3 outputs cannot be written to if they are used for special functions. When writing to Port 3, data is stored in the output register.

Port 3 is read by specifying P3 as the source register of an instruction. When reading from Port 3, the data returned is both the data on the input pins and in the output register.

5.5.3 Special Functions

Special functions for Port 3 are defined by programming the Port 3 Mode Register. By writing 0s in bit 6 through bit 1, lines P37-P30 are configured as input/output pairs (Figure 5-15). Table 5-1 shows available functions for Port 3. The special functions indicated in the figure are discussed in detail in their corresponding sections in this manual.

Port 3 input lines P33-P30 always function as interrupt requests regardless of the configuration specified in the Port 3 Mode Register.



Register F7H
Port 3 Mode Register
(Write-Only)
D7
D6
D5
D4
D3
D2
D1
D0
0 Port 2 Open-Drain
1 Port 2 Push-Pull
0 P31, P32 Digital Mode
1 P31, P32 Analog Mode
0 P32 = Input P35 = Output
1 P32 = /DAV/RDY2 P35 = RDY//DAV0
00 P33 = Input P34 = Output
01 P33 = Input P34 = /DM
10 P33 = Input P34 = /DM
11 P33 = /DAV1/RDY1 P34 = RDY1//DAV1
0 P31 = Input P36 = Output
1 P32 = /DAV2/RDY2 P36 = RDY2//DAV2
0 P30 = Input P37 = Output
1 P30 = Serial In P37 = Serial Out
0 Parity ON
1 Parity OFF

Figure 5-15. Port 3 Mode Register Configuration

Table 5-1. Port 3 Line Functions

Function	Line	Signal
Inputs	P30	Input
	P31	Input
	P32	Input
	P33	Input
Outputs	P34	Output
	P35	Output
	P36	Output
	P37	Output
Port 0 Handshake Input	P32	/DAV0/RDY0
Port 1 Handshake Input	P33	/DAV1/RDY1
Port 2 Handshake Input	P31	/DAV2/RDY2
Port 0 Handshake Output	P35	RDY0//DAV0
Port 1 Handshake Output	P34	RDY1//DAV1
Port 2 Handshake Output	P36	RDY2//DAV2
Analog Comparator Input	P31	AN1
	P32	AN2
	P33	REF
Analog Comparator Output	P34	AN1-OUT
	P35	AN2-OUT
	P37	AN2-OUT
Interrupt Requests	P30	IRQ3
	P31	IRQ2
	P32	IRQ0
	P33	IRQ1
Serial Input	P20	DI
Serial Output	P27	DO
SPI Slave Select	P35	SS
SPI Clock	P34	SK
Counter/Timer	P31	T_{IN}
	P36	T_{OUT}
External Memory Status	P34	/DM

5.6 PORT HANDSHAKE

When Ports 0, 1, and 2 are configured for handshake operation, a pair of lines from Port 3 are used for handshake controls. The handshake controls are interlocked to properly time asynchronous data transfers between the Z8® and a peripheral. One control line (/DAV) functions as a strobe from the sender to indicate to the receiver that data is available. The second control line (RDY) acknowledges receipt of the sender's data, and indicates when the receiver is ready to accept another data transfer.

In the input mode, data is latched into the Port's input register by the first /DAV signal, and is protected from being overwritten if additional pulses occur on the /DAV line. This overwrite protection is maintained until the port data is read. In the output mode, data written to the port is not protected and can be overwritten by the Z8 during the handshake sequence. To avoid losing data, the software must not overwrite the port until the corresponding interrupt request indicates that the external device has latched the data.

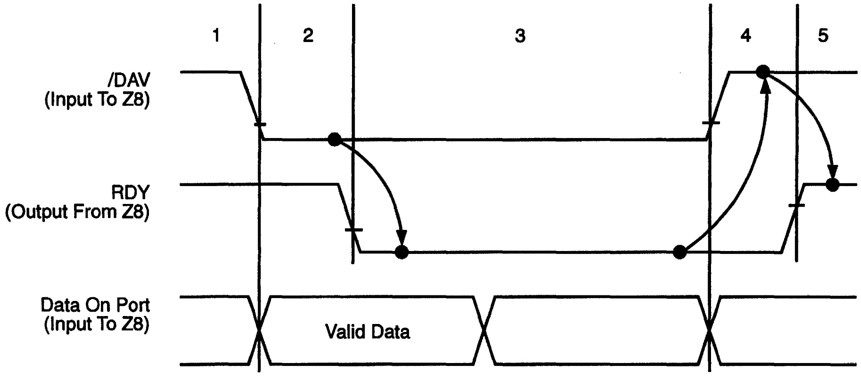
The software can always read Port 3 output and input handshake lines, but cannot write to the output handshake line.

The following is the recommended setup sequence when configuring a Port for handshake operation for the first time after a reset:

- Load P01M or P2M to configure the port for input/ output.
- Load P3 to set the Output Handshake bit to a logic 1.
- Load P3M to select the Handshake Mode for the port.

Once a data transfer begins, the configuration of the handshake lines should not be changed until the handshake is completed.

Figures 5-16 and 5-17 show detailed operation for the handshake sequence.



State 1. Port 3 Ready output is High, indicating that the Z8 is ready to accept data.

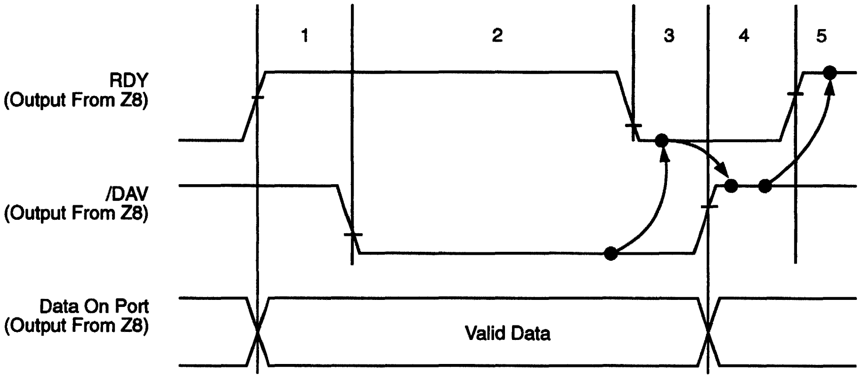
State 2. The I/O device puts data on the port and then activates the /DAV input. This causes the data to be latched into the port input register and generates an interrupt request.

State 3. The Z8 forces the Ready (RDY) output Low, signaling to the I/O device that the data has been latched.

State 4. The I/O device returns the /DAV line High in response to the RDY going Low.

State 5. The Z8® software must respond to the interrupt request and read the contents of the port in order for the handshake sequence to be completed. The RDY line goes High if and only if the port has not been read and /DAV is High. This returns the interface to its initial state.

Figure 5-16. Z8 Input Handshake



State 1. RDY input is High indicating that the I/O device is ready to accept data.

State 2. The Z8® writes to the port register to initiate a data transfer. Writing the port outputs new data and forces /DAV Low if and only if RDY is High.

State 3. The I/O device forces RDY Low after latching the data RDY Low causes an interrupt request to be generated. The Z8 can write new data in response to RDY going Low; however, the data is not output until State 5.

State 4. The /DAV output from the Z8 is driven High in response to RDY going Low.

State 5. The /DAV goes High, the I/O device is free to raise RDY High thus returning the interface to its initial state.

Figure 5-17. Z8 Output Handshake

In applications requiring a strobed signal instead of the interlocked handshake, the Z8® can satisfy this requirement as follows:

- In the Strobed Input mode, data can be latched in the Port input register using the /DAV input. The data transfer rate must allow enough time for the software to read the Port before strobing in the next character. The RDY output is ignored.
- In the Strobed Output Mode, the RDY input should be tied to the /DAV output.

Figures 5-18 and 5-19 illustrate the strobed handshake connections.

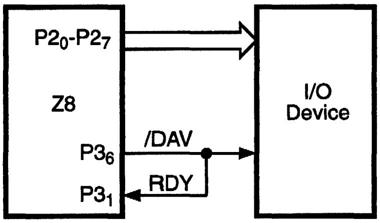


Figure 5-18. Output Strobed Handshake on Port 2

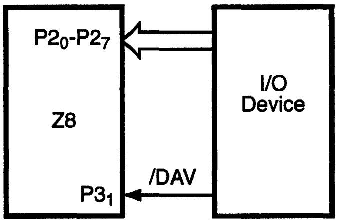


Figure 5-19. Input Strobed Handshake on Port 2

5.7 I/O PORT RESET CONDITIONS

5.7.1 Full Reset

After a hardware reset, Watch-Dog Timer (WDT) reset, or a Power-On Reset (POR), Port Mode Registers P01M, P2M, and P3M are set as shown in Figures 5-20 through 5-22. Port 2 is configured for input operation on all bits and is set for open-drain (Figure 5-22). If push-pull outputs are desired for Port 2 outputs, remember to configure them using P3M. Please note that a WDT time-out from Stop-Mode Recovery does not do a full reset. Certain registers that are not reset after Stop-Mode Recovery will not be reset.

For the condition of the Ports after Stop-Mode Recovery, please refer to specific device product specifications. In some cases, the Z8® has the P01M, P2M, and P3M control register set back to the default condition after reset while others do not.

All special I/O functions of Port 3 are inactive, with P33-P30 set as inputs and P37-P34 set as outputs (Figure 5-22).

Note: Because the types and amounts of I/O vary greatly among the Z8 family devices, the user is advised to review the selected device's product specifications for the register default state after reset.

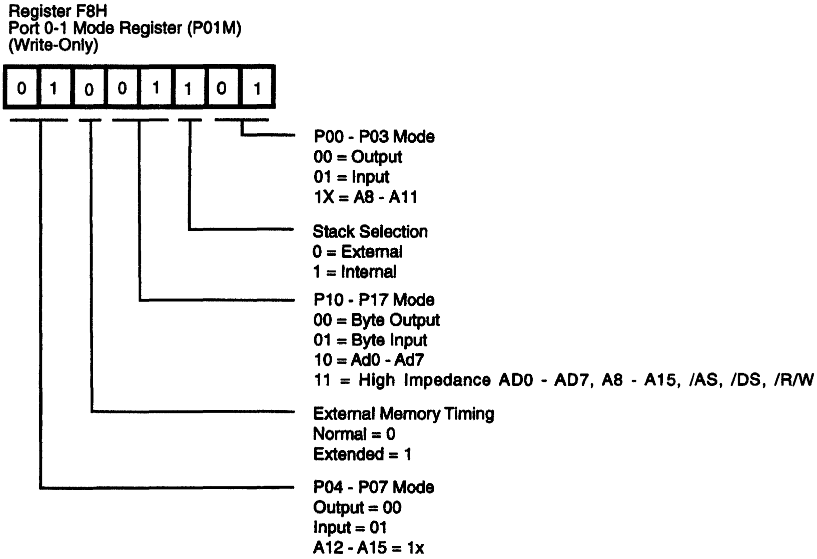


Figure 5-20. Port 0/1 Reset

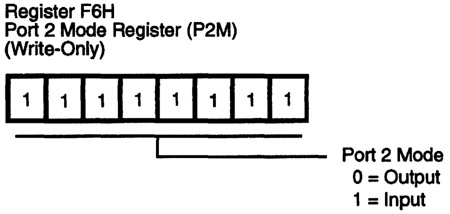


Figure 5-21. Port 2 Reset

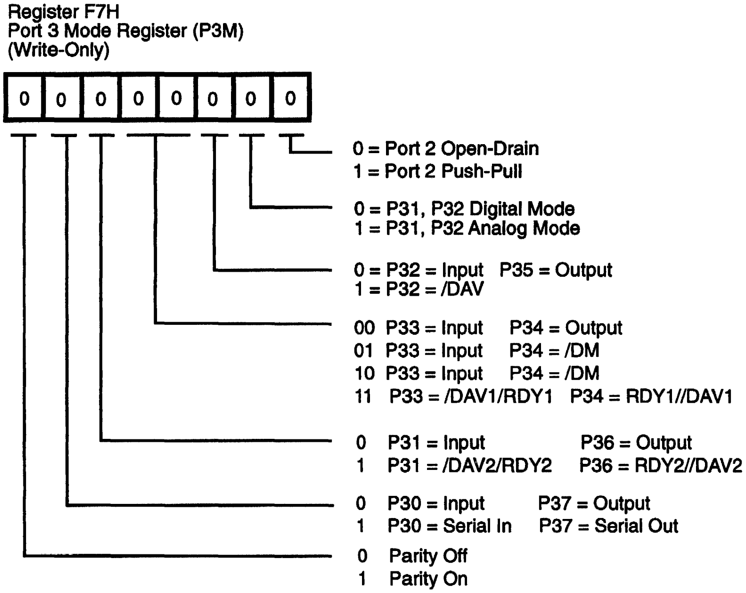

Register F7H
Port 3 Mode Register (P3M)
(Write-Only)
0 0 0 0 0 0 0 0
0 = Port 2 Open-Drain
1 = Port 2 Push-Pull
0 = P31, P32 Digital Mode
1 = P31, P32 Analog Mode
0 = P32 = Input P35 = Output
1 = P32 = /DAV
00 P33 = Input P34 = Output
01 P33 = Input P34 = /DM
10 P33 = Input P34 = /DM
11 P33 = /DAV1/RDY1 P34 = RDY1//DAV1
0 P31 = Input P36 = Output
1 P31 = /DAV2/RDY2 P36 = RDY2//DAV2
0 P30 = Input P37 = Output
1 P30 = Serial In P37 = Serial Out
0 Parity Off
1 Parity On

Figure 5-22. Port 3 Mode Reset

5.8 ANALOG COMPARATORS

Select Z8® devices include two independent on-chip analog comparators. See the device product specification for feature availability and use. Port 3, Pins P31 and P32 each have a comparator front end. The comparator reference voltage, pin P33, is common to both comparators. In Analog Mode, the P31 and P32 are the positive inputs to the comparators and P33 is the reference voltage supplied to both comparators. In Digital Mode, pin P33 can be used as a P33 register input or IRQ1 source. P34, P35, or P37 may output the comparator outputs by software-programming the PCON Register bit D0 to 1.

5.8.1 Comparator Description

Two on-board comparators can process analog signals on P31 and P32 with reference to the voltage on P33. The analog function is enabled by programming the Port 3 Mode Register (P3M bit 1). For interrupt functions during analog mode, P31 and P32 can be programmable as rising, falling, or both edge triggered interrupts (IRQ register bits 6 and bit 7).

Note: P33 cannot generate an external interrupt while in this mode. P33 can only generate interrupts in the Digital Mode.

Note: Port 3 inputs must be in digital mode if Port 3 is a Stop-Mode Recovery source. The analog comparator is disabled in STOP mode.

P31 can be used as T_{IN} in Analog or Digital Modes, but it must be referenced to P33, when in Analog Mode.

Register F7H
Port 3 Mode Register (P3M)
(Write-Only)

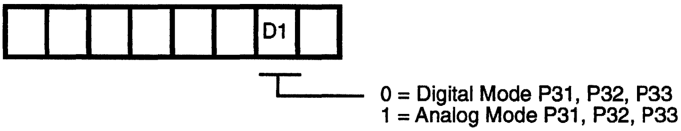


Figure 5-23. Port 3 Input Analog Selection

ERF Bank F
Register 00H
Port Configuration Register (PCON)
(Write-Only)

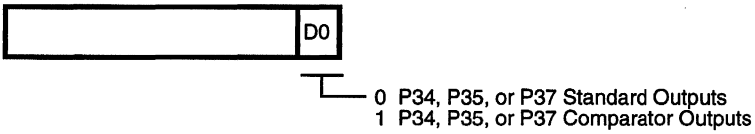

D0
0 P34, P35, or P37 Standard Outputs
1 P34, P35, or P37 Comparator Outputs

Figure 5-24. Port 3 Comparator Output Selection

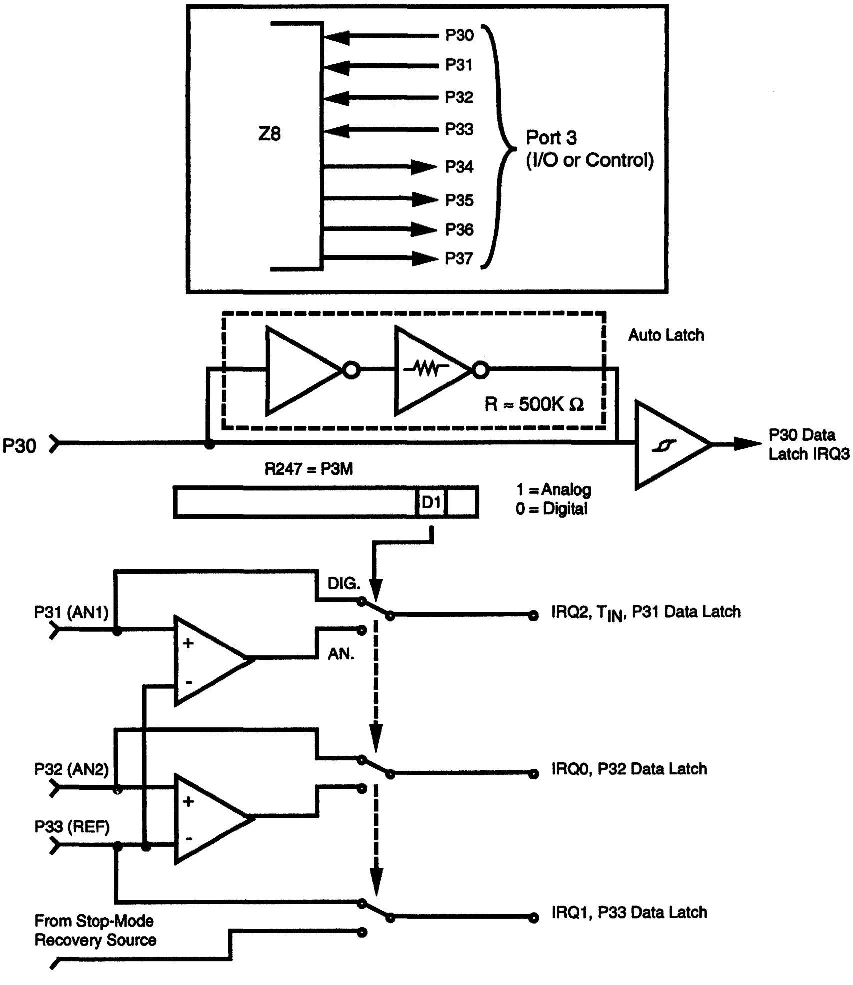


Figure 5-25. Port Configuration of Comparator Inputs on P31, P32, and P33

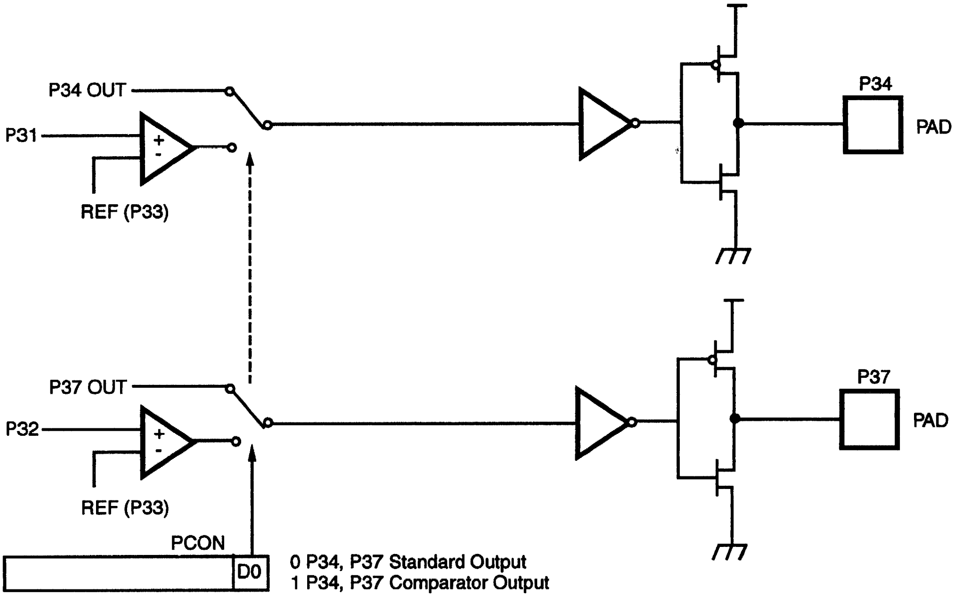


Figure 5-26. Port 3 Configuration

5.8.2 Comparator Programming

Example of enabling analog comparator mode.

```
LD P3M, #XXXX XX1XB
```

Note: X = don't care.

Example of enabling analog comparator output.

```
LD RP, #%0FH           ;Sets register pointer to
                       ;working register group 0
                       ;and Expanded Register
                       ;File bank.

LD R0, #XXXX XXX1B     ;Enables comparator
                       ;outputs using PCOM
                       ;Register programming.
```

5.8.3 COMPARATOR OPERATION

After enabling the Analog Comparator mode, P33 becomes a common reference input for both comparators. The P33 (Ref) is hard wired to the reference inputs to both comparators and cannot be separated. P31 and P32 are always connected to the positive inputs to the comparators. P31 is the positive input to comparator AN1 while P32 is the positive input to comparator AN2. The outputs to comparators AN1 and AN2 are AN1-out and AN2-out, respectively.

The comparator output reflects the relationship between the positive input to the reference input.

Example: If the voltage on AN1 is higher than the voltage on Ref then AN1-out will be at a high state. If voltage on AN2 is lower than the voltage on Ref then AN2-out will be at a Low state. In this example, when the Port 3 register is read, Bits D1 = 1 and D2 = 0. If the comparator outputs are enabled to come out on P34 and P37, then P34 = 1 and P37 = 0. Please note that the previous data stored in P34 and P37 is not disturbed. Once the comparator outputs are deselected the stored values in the P34 and P37 register bits will be reflected on these pins again.

5.8.4 Interrupts

In the example from Section 5.8.3, P32 (AN2) will generate an interrupt based on the result of the comparison being low and the Interrupt Request Register (IRQ FAH) having bits D7=0 and D6=0. If IRQ D7=1 and D6=0 then both P31 and P32 would generate interrupts.

5.8.5. Comparator Definitions

5.8.5.1. V_{ICR}

The usable voltage range for both positive inputs and the reference input is called the common mode voltage range (V_{ICR}). The comparator is not guaranteed to work if the inputs are outside of the V_{ICR} range.

5.8.5.2. V_{OFFSET}

The absolute value of the voltage between the positive input and the reference input required to make the comparator output voltage switch is the input offset voltage (Voffset). If AN1 is 3.000V and Ref is 3.001V when the comparator output switches states then the Voffset = 1mV.

5.8.5.3. I_{IO}

For CMOS voltage comparator inputs, the input offset current (I_{IO}) is the leakage current of the CMOS input gate.

5.8.6. RUN Mode

P33 is not available as an interrupt input during Analog Mode. P31 and P32 are valid interrupt inputs in conjunction with P33 (Ref) when in the Analog Mode.

P31 can still be used as T_{IN} when the analog mode is selected. If comparator outputs are desired to be outputted on the Port 3 outputs, please refer to specific products specification for priority of muxing when other special features are sharing those same Port 3 pins.

5.8.7. HALT Mode

The analog comparators are functional during HALT Mode if the Analog Mode has been enabled. P31 and P32, in conjunction with P33 (Ref) will be able to generate interrupts. Only P33 cannot generate an interrupt since the P33 input goes directly to the Ref input of the comparators and is disconnected from the interrupt sensing circuits.

5.8.8. STOP Mode

The analog comparators are disabled during STOP Mode so it does not use any current at that time. If P31, P32, or P33 are used as a source for Stop-Mode Recovery, the Port 3 Digital Mode must be selected by setting bit D1=0 in the Port 3 Mode Register. Otherwise in STOP Mode, the P31, P32, and P33 cannot be sensed. If the Analog Mode was selected when entering STOP Mode, it will still be enabled after a valid SMR triggered reset.

5.9 OPEN-DRAIN CONFIGURATION

All Z8s can configure Port 2 to provide open-drain outputs by programming the Port 3 Mode Register (P3M) bit D0=0.

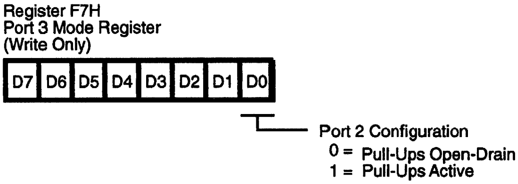


Figure 5-27. Port 2 Configuration

Other Z8s that have a Port Configuration Register (PCON) that can configure Port 0 and Port 1 to provide open-drain outputs. The PCON Register is located in Expanded Register File (ERF) Bank F, Register 00H. See Figure 5-28.

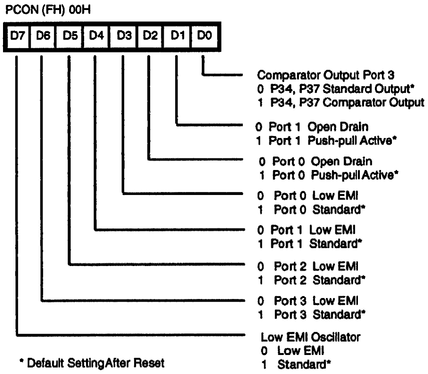


Figure 5-28. Port Configuration Register (PCON) (Write-Only)

Port 1 Open-Drain (D1). Port 1 can be configured as open-drain by resetting this bit (D1=0) or configured as push-pull active by setting this bit (D1=1). The default value is 1.

Port 0 Open Drain (D2). Port 0 can be configured as open-drain by resetting this bit (D2=0) or configured as push-pull active by setting this bit (D2=1). The default value is 1.

5.10 LOW EMI EMISSION

Some Z8s can be programmed to operate in a Low EMI Emission Mode using the Port configuration register (PCON). The PCON register allows the oscillator and all I/O ports to be programmed in the Low-EMI Mode independently. Other Z8s may offer a ROM Mask or OTP programming option to configure the Z8 Ports and oscillator globally to a Low-EMI mode (where the XTAL frequency is set equal to the internal system clock frequency.

Use of the Low EMI feature results in:

- The output pre-drivers slew rate reduced to 10 ns (typical).
- Low EMI output drivers have resistance of 200 Ohms (typical).
- Low EMI Oscillator.
- All output drivers are approximately 25 percent of the standard drive.
- Internal SCLK/TCLK = XTAL operation limited to a maximum of 4 MHz - 250 ns cycle time, when Low EMI Oscillator is selected and system clock (SCLK=XTAL, SMR Reg. Bit D1=1).

For Z8s having the PCON register feature, the following bits control the Low EMI options:

- **Low EMI Port 0 (D3).** Port 0 can be configured as a Low EMI Port by resetting this bit (D3=0) or configured as a Standard Port by setting this bit (D3=1). The default value is 1.

- **Low EMI Port 1 (D4).** Port 1 can be configured as a Low EMI Port by resetting this bit (D4=0) or configured as a Standard Port by setting this bit (D4=1). The default value is 1.

- **Low EMI Port 2 (D5).** Port 2 can be configured as a Low EMI Port by resetting this bit (D5=0) or configured as a Standard Port by setting this bit (D5=1). The default value is 1.

- **Low EMI Port 3 (D6).** Port 3 can be configured as a Low EMI Port by resetting this bit (D6=0) or configured as a Standard Port by setting this bit (D6=1). The default value is 1.

- **Low EMI OSC (D7).** This bit of the PCON Register controls the Low EMI oscillator. A 1 in this location configures the oscillator with standard drive, while a 0 configures the oscillator with low noise drive. The Low-EMI mode will reduce the drive of the oscillator (OSC). The default value is 1. XTAL/2 mode is not effected by this bit.

Note: The maximum external clock frequency is 4 MHz when running in the Low EMI oscillator mode.

Please refer to the selected device product specification for availability of the Low EMI feature and programming options.

5.11 INPUT PROTECTION

All CMOS ROM Z8s have I/O pins with diode input protection. There is a diode from the I/O pad to V_{CC} and to V_{SS}. See Figure 5-29A.

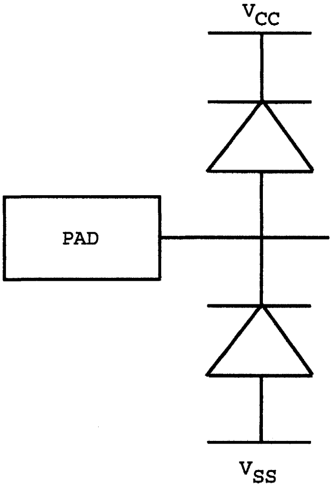


Figure 5-29a. Diode Input Protection

On CMOS OTP EPROM Z8's, the Port 3 inputs P31, P32, P33 and the XTAL 1 pin have only the input protection diode from pad to V_{SS}. See Figure 5-29B.

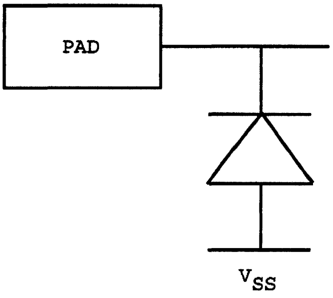


Figure 5-29b. OTP Diode Input Protection

The high-side input protection diodes were removed on these pins to allow the application of +12.5V during the various OTP programming modes.

For better noise immunity in applications that are exposed to system EMI, a clamping diode to V_{CC} from these pins may be required to prevent entering the OTP programming mode or to prevent high voltage from damaging these pins.

5.12. CMOS Z8 AUTO LATCHES

I/O port bits that are configurable as inputs are protected against open circuit conditions using Auto Latches. An Auto Latch is a circuit which, in the event of an open circuit condition, latches the input at a valid CMOS level. This inhibits the tendency of the input transistors to self-bias in the forward active region, thus drawing excessive supply current. A simplified schematic of the CMOS Z8 I/O circuit is shown in Figure 5-30.

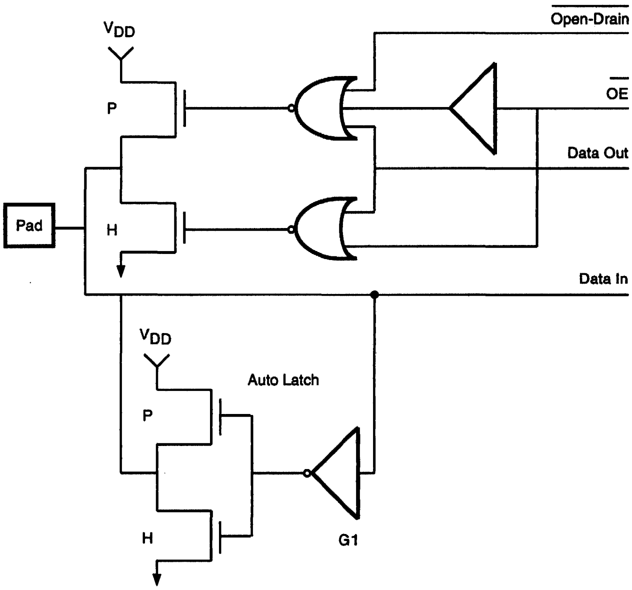


Figure 5-30. Simplified CMOS Z8 I/O Circuit

The operation of the Auto Latch circuit is straight-forward. Assume the input pad is latched at +5V (logic 1). The inverter G1 inverts the bit, turning the P-channel FET ON and the N-channel FET OFF. The output of the circuit is effectively shorted to V_{DD}, returning +5V to the input. If the pad is then disconnected from the +5V source, the Auto Latch will hold the input at the previous state. If the device is powered up with the input floating, the state of the Auto Latch will be at either supply, but which state is unpredictable.

There are four operating conditions which will activate the Auto Latches. The first, which occurs when the input pin is physically disconnected from any source, is the most obvious. The second occurs when the input is connected to the output of a device with tri-state capability.

The Auto Latch will also activate when the input voltage at the pin is not within 200 microV or so of either supply rail. In this case, the circuit will draw current, which is not significant compared to the Icc operating current of the device, but will increase I_{CC2} STOP Mode current of the device dramatically.

The fourth condition occurs when the I/O bit is configured as an output. Referring to the output section of Figure 5-30, there are two ways of tri-stating the port pin. The first is by configuring the port as an input, which disables the /OE signal turning both transistors off. The second can be achieved in output mode by writing a "1" to the output port, then activating the open drain mode. Both transistors are again off, and the port bit is in a high impedance state. The Auto Latches then pull the input section toward V_{DD}.

Auto Latch Model:

The Auto Latch's equivalent circuit is shown in Figure 5-31. When the input is high, the circuit consists of a resistance Rp from V_{DD} (the P-channel transistor in its ON state) and a much greater resistance Rh to GND. Current Iao flows from V_{DD} to the output. When the input is low, the circuit may be modeled as a resistance Rp from GND (the N-channel transistor in the ON state) and a much greater resistance Rh to V_{DD}. Current Iao now flows from the input to ground. The Auto Latch is characterized with respect to Iao, so the equivalent resistance Rp is calculated according to RP= $(V_{DD}-V_{IN})$/Iao. The worst case equivalent resistance Rp (min) may be calculated at the worst case input voltage, Vi= Vih(min).

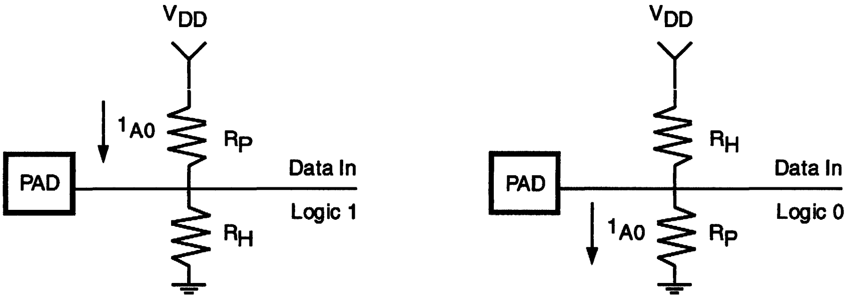


Figure 5-31. Auto Latch Equivalent Circuit

Design Considerations:

For circuits in which the Auto Latch is active, consideration should be given to the loading constraints of the Auto Latches. For example, with weak values of V_{IN}, close to Vih (min) or Vil (max), pullup or pull-down resistances must be calculated using Ref = R/Rp. For best case STOP mode operation, the inputs should be within 200 mV of the supply rails.

In output mode, if a port bit is forced into a tri-state condition, the Auto Latches will force the pad to V_{DD}. If there is an external pulldown resistor on the pin, the voltage at the pin may not switch to GND due to the Auto Latch. As shown in Figure 5-32, the equivalent resistance of the Auto Latch and the external pulldown form a voltage divider, and if the external resistor is large, the voltage developed across it will exceed Vil(max). For worst case:

Vil(max > V_{DD} [Rext/(Rext+Rp)]
Rext(max) = [(Vil(max)/V_{DD})Rp]/[1-(Vil(max)/V_{DD})]

For V_{DD} = 5.0V and Iao = 5 uA we have Vih(max) =0.8V:
Rext(max) = (0.16/1M)/(1-0.16) = 190 K ohms.

Rp increases rapidly with V_{DD}, so increased V_{DD} will relax the requirement on Rext.

In summary, the CMOS Z8 Auto Latch inhibits excessive current drain in Z8 devices by latching an open input to either V_{DD} or GND. The effect of the Auto Latch on the I/O characteristics of the device may be modeled by a current Iao and a resistor Rp, whose value is V_{DD}/Iao.

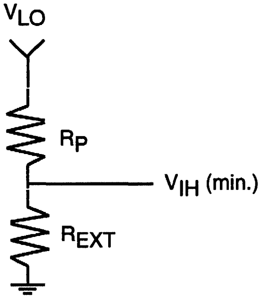


Figure 5-32. Effect of Pulldown Resistors on Auto Latches

CHAPTER 6
COUNTER/TIMERS

6.1 INTRODUCTION

The Z8® provides up to two 8-bit counter/timers, T0 and T1, each driven by its own 6-bit prescaler, PRE0 and PRE1 (Figure 6-1). Both counter/timers are independent of the processor instruction sequence, that relieves software from time-critical operations such as interval timing or event counting. Some MCUs offer clock scaling using the SMR register. See the device product specification for clock available options. The following description is typical.

Each counter/timer operates in either Single-Pass or Continuous mode. At the end-of-count, counting either stops or the initial value is reloaded and counting continues. Under software control, new values are loaded immediately or when the end-of-count is reached. Software also controls the counting mode, how a counter/timer is started or stopped, and its use of I/O lines. Both the counter and prescaler registers can be altered while the counter/timer is running.

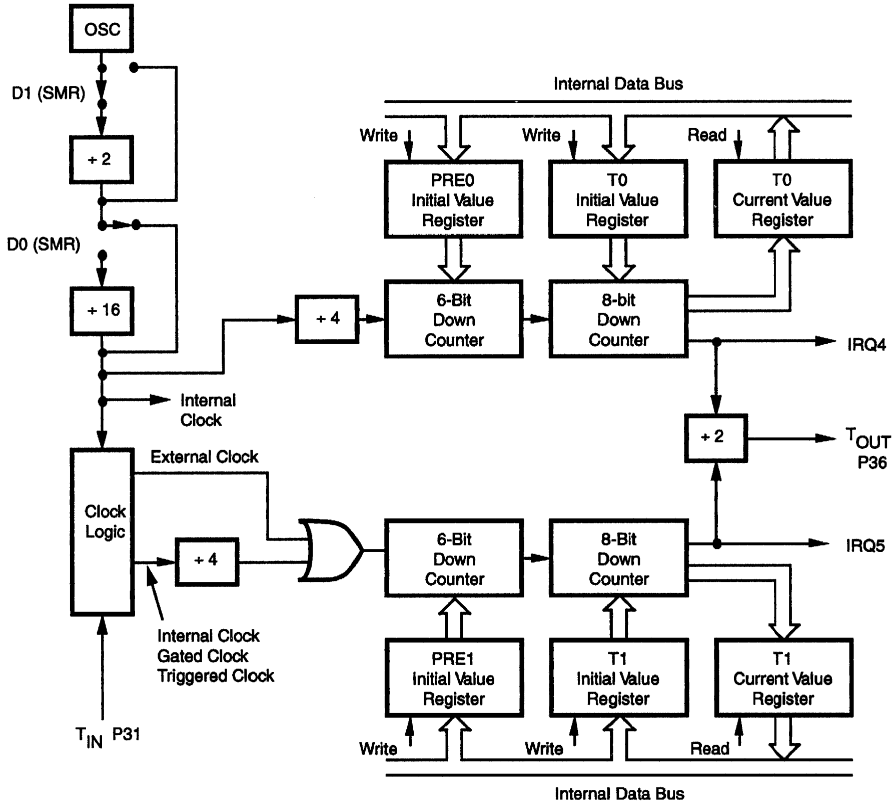


Figure 6-1. Counter/Timer Block Diagram

Counter/timers 0 and 1 are driven by a timer clock generated by dividing the internal clock by four. The divide-by-four stage, the 6-bit prescaler, and the 8-bit counter/timer form a synchronous 16-bit divide chain. Counter/timer 1 can also be driven by a external input (T_{IN}) using P31. Port 3 line P36 can serve as a timer output (T_{OUT}) through which T0, T1, or the internal clock can be output. The timer output will toggle at the end-of-count.

The counter/timer, prescaler, and associated mode registers are mapped into the register file as shown in Figure 6-2. This allows the software to treat the counter/timers as general-purpose registers, and eliminates the need for special instructions.

6.2 PRESCALERS AND COUNTER/TIMERS

The prescalers, PRE0 (F5H) and PRE1 (F3H), each consist of an 8-bit register and a 6-bit down-counter as shown in Figure 6-1. The prescaler registers are write-only registers. Reading the prescalers returns the value FFH. Figures 6-3 and 6-4 show the prescaler registers.

The six most significant bits (D2-D7) of PRE0 or PRE1 hold the prescalers count modulo, a value from 1 to 64 decimal. The prescaler registers also contain control bits that specify T0 and T1 counting modes. These bits also indicate whether the clock source for T1 is internal or external. These control bits will be discussed in detail throughout this chapter.

The counter/timer registers, T0 (F4H) and T1 (F2H), each consist of an 8-bit down-counter, a write-only register that holds the initial count value, and a read-only register that holds the current count value (Figure 6-1). The initial value can range from 1 to 256 decimal (01H,02H,..,00H). Figure 6-5 illustrates the counter/timer registers.

DEC		HEX Identifiers
247	Port 3 Mode	F7
245	To Prescaler	F5
244	Timer/Counter0	F4
243	T1 Prescaler	F3
242	Timer/Counter1	F2
241	Timer Mode	F1

Figure 6-2. Counter/Timer Register Map

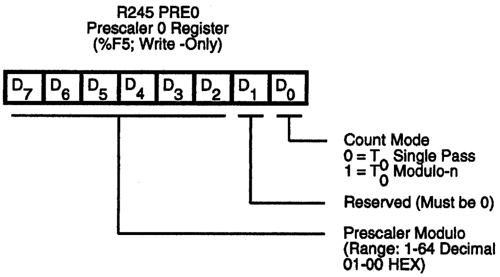


Figure 6-3. Prescaler 0 Register

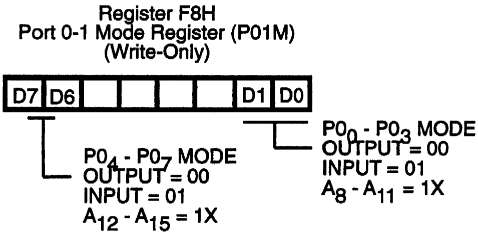


Figure 6-4. Prescaler 1 Register

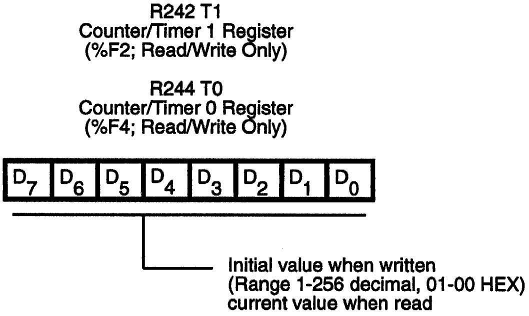


Figure 6-5. Counter / Timer 0 and 1 Registers

6.3 COUNTER/TIMER OPERATION

Under software control, counter/timers are started and stopped via the Timer Mode Register (TMR,F1H) bits D_0-D_3 (Figure 6-6). Each counter/timer is associated with a Load bit and an Enable Count bit.

6.3.1 Load and Enable Count Bits

Setting the Load bit (D_0 for T0 and D_2 for T1) transfers the initial value in the prescaler and the counter/timer registers into their respective down-counters. The next internal clock resets bits D_0 and D_2 to 0, readying the Load bit for the next load operation. New values may be loaded into the down-counters at any time. If the counter/timer is running, it continues to do so and starts the count over with the new value. Therefore, the Load bit actually functions as a software re-trigger.

The counter timers remain at rest as long as the Enable Count bits are 0. To enable counting, the Enable Count bit (D_1 for T0 and D_3 for T1) must be set to 1. Counting actually starts when the Enable Count bit is written by an instruction. The first decrement occurs four internal clock periods after the Enable Count bit has been set. If T1 is configured to use an external clock, the first decrement begins on the next clock period. The Load and Enable Count bits can be set at the same time. For example, using the instruction:

OR TMR,#03H

sets both D0 and D1 of the TMR. This loads the initial values of PRE0 and T0 into their respective counters and starts the count after the M2T2 machine state after the operand is fetched (Figure 6-7).

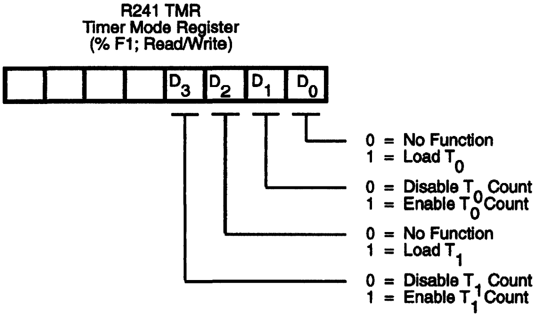


Figure 6-6. Timer Mode Register

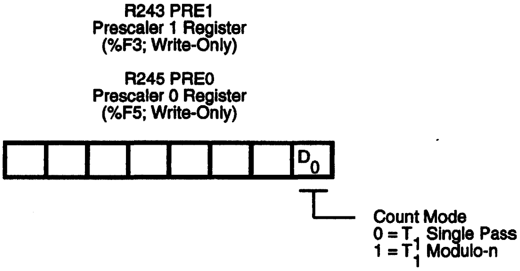


Figure 6-7. Starting The Count

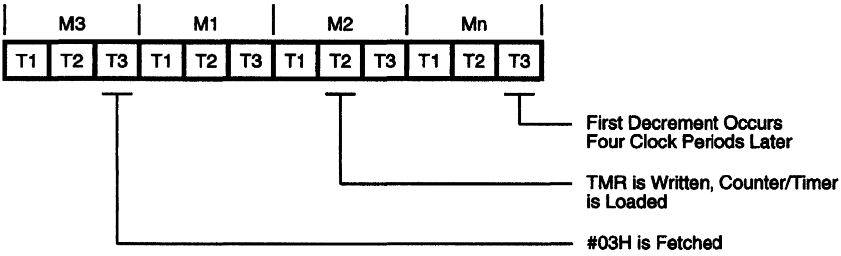


Figure 6-8. Counting Modes

6.3.2 Prescaler Operations

During counting, the programmed clock source drives the 6-bit Prescaler Counter. The counter is counted down from the value specified by bits of the corresponding Prescaler Register, PRE0 (bit 7 to bit 2) or PRE1 (bit 7 to bit 2). (Figures 6-3, 6-4). When the Prescaler Counter reaches its end-of-count, the initial value is reloaded and counting continues. The prescaler never actually reaches 0. For example, if the prescaler is set to divide-by-three, the count sequence is:

3-2-1-3-2-1-3-2-1-3...

Each time the prescaler reaches its end of count a carry is generated, that allows the Counter/Timer to decrement by one on the next timer clock input. When the Counter/Timer and the prescaler both reach the end-of-count, an interrupt request is generated (IRQ4 for T0, IRQ5 for T1). Depending on the counting mode selected, the Counter/Timer will either come to rest with its value at 00H (Single-Pass Mode) or the initial value will be automatically reloaded and counting will continue (Continuous Mode). The counting modes are controlled by bit 0 of PRE0 and bit 0 of PRE1. (Figure 6-8). A 0, written to this bit configures the counter for Single-pass counting mode, while a 1 written to this bit configures the counter for Continuous mode.

The Counter/Timer can be stopped at any time by setting the Enable Count bit to 0, and restarted by setting it back to 1. The Counter/Timer will continue its count value at the time it was stopped. The current value in the Counter/Timer can be read at any time without affecting the counting operation.

Note: The prescaler registers are write-only and cannot be read.

New initial values can be written to the prescaler or the Counter/Timer registers at any time. These values will be transferred to their respective down counters on the next load operation. If the Counter/Timer mode is Continuous, the next load occurs on the timer clock following an end-of-count. New initial values should be written before the desired load operation, since the prescalers always effectively operate in Continuous count mode.

The time interval (i) until end-of-count, is given by the equation:

$$i = t X p X v$$

in which:

t = four divided by the internal clock frequency.

The internal clock frequency defaults to the external clock source (XTAL, ceramic resonator, and others) divided by 2. Some Z8® microcontrollers allow this divisor to be changed via the Stop-Mode Recovery register. See the product data sheet for available clock divisor options.

Note that t is equal to eight divided-by-XTAL frequency of the external clock source for T1 (external clock mode only).

p = the prescaler value (1 - 63) for T_0 and T_1.

The minimum prescaler count of 1 is achieved by loading 000001xx. The maximum prescaler count of 63 is achieved by loading 111111xx.

v = the Counter/Timer value (1 - 256)

Minimum duration is achieved by loading 01H (1 prescaler output count), maximum duration is achieved by loading 00H (256 prescaler outputs counts).

It should be apparent the prescaler and counter/timer are true divide-by-n counters.

6.4 T_{OUT} Modes

The Timer Mode Register TMR (F1H) (Figure 6-9), is used in conjunction with the Port 3 Mode Register P3M (F7H) (Figure 6-10) to configure P36 for T_{OUT} operation for T0 and T1. In order for T_{OUT} to function, P36 must be defined as an output line by setting P3M bit 5 to 0. Output is controlled by one of the counter/timers (T0 or T1) or the internal clock.

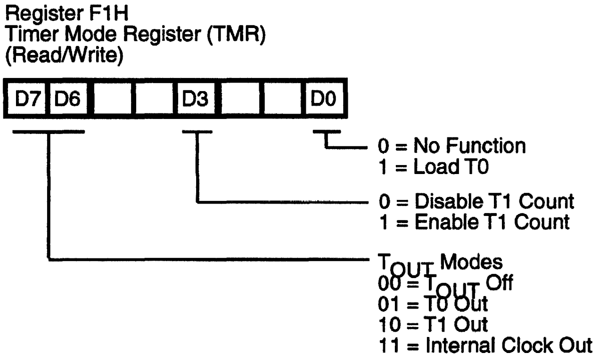


Figure 6-9. Timer Mode Register (T_{OUT} Operation)

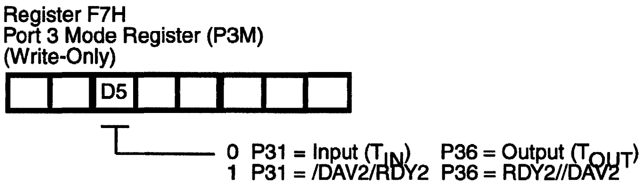

Register F7H
Port 3 Mode Register (P3M)
(Write-Only)
D5
0 P31 = Input (TIN) P36 = Output (TOUT)
1 P31 = /DAV2/RDY2 P36 = RDY2//DAV2

Figure 6-10. Port 3 Mode Register (T_{OUT} Operation)

The counter/timer to be output is selected by TMR bit 7 and bit 6. T0 is selected to drive the T_{OUT} line by setting bit) 7 to 0 and bit 6 to 1. Likewise, T1 is selected by setting bit 7 and bit 6 to 1 and 0 , respectively. The counter/timer T_{OUT} mode is turned off by setting TMR bit and bit 6 both to 0, freeing P36 to be a data output line.

T_{OUT} is initialized to a logic 1 whenever the TMR Load bit (bit 0 for T0 or bit 1 for T1) is set to 1. The T_{OUT} configuration timer load, and Timer Enable Count bits for the counter/timer driving the T_{OUT} pin can be set at the same time. For example, using the instruction:

```
OR TMR,#43H
```

- Configures T0 to drive the T_{OUT} pin (P36).
- Sets the P36 Tout pin to a logic 1 level.
- Loads the initial PRE0 and T0 levels into their respective counters and starts the counter after the M2T2 machine state after the operand is fetched.

At end-of-count, the interrupt request line (IRQ4 or IRQ5), clocks a toggle flip-flop. The output of this flip-flop drives the T_{OUT} line, P36. In all cases, when the selected counter/timer reaches its end-of-count, T_{OUT} toggles to its opposite state (Figure 6-11). If, for example, the counter/timer is in Continuous Counting Mode, Tout will have a 50 percent duty cycle output. This duty cycle can easily be controlled by varying the initial values after each end-of-count.

The internal clock can be selected as output instead of T0 or T1 by setting TMR bit 7 and bit 6 both to 1. The internal clock (XTAL frequency/2) is then directly output on P36 (Figure 6-12).

While programmed as T_{OUT}, P36 cannot be modified by a write to port register P3. However, the Z8® software can examine the P36 current output by reading the port register.

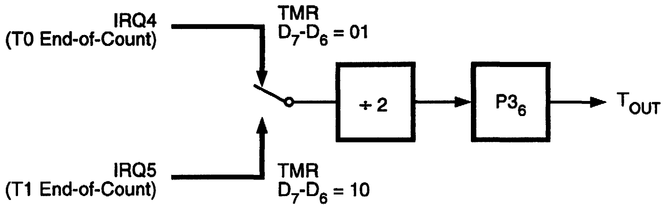


Figure 6-11. T0 and T1 Output Through T_{OUT}

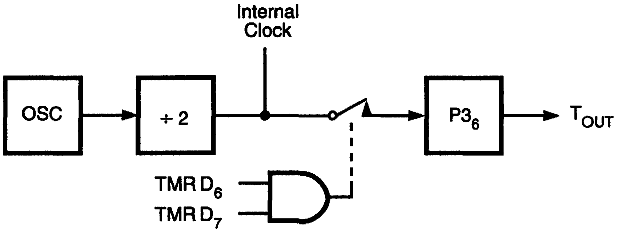


Figure 6-12. Internal Clock Output Through T_{OUT}

6.5 T_{IN} MODES

The Timer Mode Register TMR (F7H) (Figure 6-13) is used in conjunction with the Prescaler Register PRE1 (F7H) (Figure 6-14) to configure P31 as T_{IN}. T_{IN} is used in conjunction with T1 in one of four modes:

- External Clock Input
- Gated Internal Clock
- Triggered Internal Clock
- Retriggerable Internal Clock

Note: The T_{IN} mode is restricted for use with timer 1 only. To enable the T_{IN} mode selected (via TMR bits 4- 5), bit 1 of PRE1 must be set to 1.

The counter/timer clock source must be configured for external by setting the PRE1 Register bit 2 to 0. The Timer Mode Register bit 5 and bit 4 can then be used to select the desired T_{IN} operation.

For T1 to start counting as a result of a T_{IN} input, the Enable Count bit (bit 3 in TMR) must be set to 1. When using T_{IN} as an external clock or a gate input, the initial values must be loaded into the down counters by setting the Load bit (bit 2 in TMR) to a 1 before counting begins. In the descriptions of T_{IN} that follow, it is assumed the programmer has performed these operations. Initial values are automatically loaded in Trigger and Retrigger modes so software loading is unnecessary.

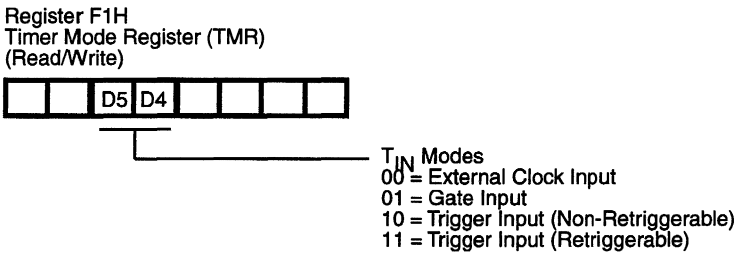

Register F1H
Timer Mode Register (TMR)
(Read/Write)
D5
D4
T_IN Modes
00 = External Clock Input
01 = Gate Input
10 = Trigger Input (Non-Retriggerable)
11 = Trigger Input (Retriggerable)

Figure 6-13. Timer Mode Register (T_{IN} Operation)

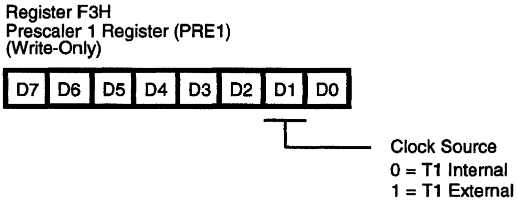


Figure 6-14. Prescaler 1 Register (T_{IN} Operation)

It is suggested that P31 be configured as an input line by setting P3M Register bit 5 to 0, although T_{IN} is still functional if P31 is configured as a handshake input .

Each High-to-Low transition on T_{IN} generates an interrupt request IRQ2, regardless of the selected T_{IN} mode or the enabled/disabled state of T1. IRQ2 must therefore be masked or enabled according to the needs of the application.

6.5.1 External Clock Input Mode

The T_{IN} External Clock Input Mode (TMR bit 5 and bit 4 both set to 0) supports counting of external events, where an event is considered to be a High-to-Low transition on T_{IN} (Figure 6-15).

Note: See the product data sheet for the minimum allowed T_{IN} external clock input period (T_P T_{IN}).

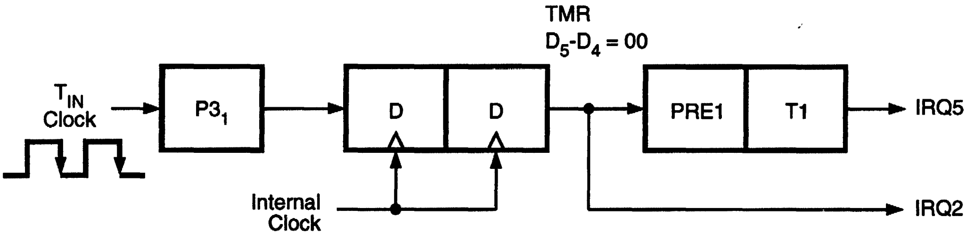


Figure 6-15. External Clock Input Mode

6.5.2 Gated Internal Clock Mode

The T_{IN} Gated Internal Clock Mode (TMR bit 5 and bit 4 set to 0 and 1 respectively) measures the duration of an external event. In this mode, the T1 prescaler is driven by the internal timer clock, gate by a High level on T_{IN} (Figure 6-16). T1 counts while T_{IN} is High and stops counting while T_{IN} is Low. Interrupt request IRQ2 is generated on the High-to-Low transition of T_{IN} signalling the end of the gate input. Interrupt request IRQ5 is generated if T1 reaches its end-of-count.

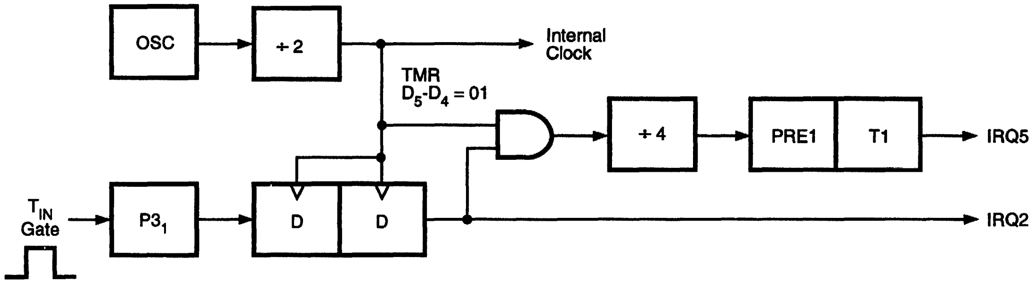


Figure 6-16. Gated Clock Input Mode

6.5.3 Triggered Input Mode

The T_{IN} Triggered Input Mode (TMR bits 5 and 4 are set to 1 and 0 respectively) causes T1 to start counting as the result of an external event (Figure 6-17). T1 is then loaded and clocked by the internal timer clock following the first High-to-Low transition on the T_{IN} input. Subsequent T_{IN} transitions do not affect T1. In the Single-Pass Mode, the Enable bit is reset whenever T1 reaches its end-of-count. Further T_{IN} transitions will have no effect on T1 until software sets the Enable Count bit again. In Continuous mode, once T1 is triggered counting continues until software resets the Enable Count bit. Interrupt request IRQ5 is generated when T1 reaches its end-of-count.

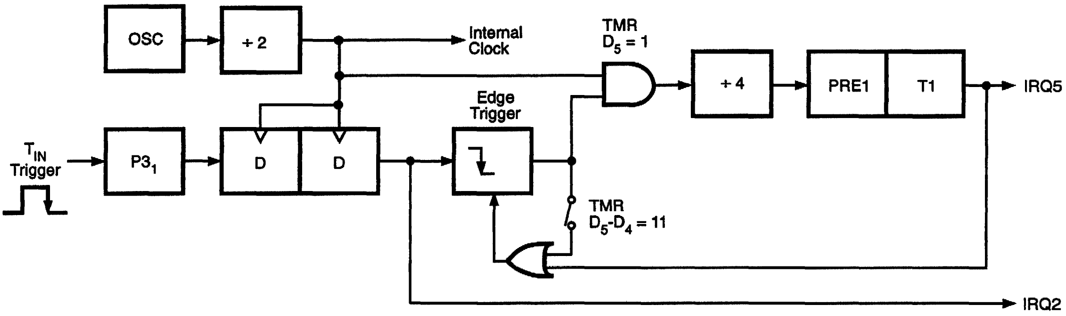


Figure 6-17. Triggered Clock Mode

6.5.4 Retriggerable Input Mode

The T_{IN} Retriggerable Input Mode (TMR bits 5 and 4 are set to 1) causes T1 to load and start counting on every occurrence of a High-to-Low transition on T_{IN} (Figure 6-17). Interrupt request IRQ5 will be generated if the programmed time interval (determined by T1 prescaler and counter/timer register initial values) has elapsed since the last High-to-Low transition on T_{IN}. In Single-Pass Mode, the end-of-count resets the Enable Count bit. Subsequent T_{IN} transitions will not cause T1 to load and start counting until software sets the Enable Count bit again. In Continuous Mode, counting continues once T1 is triggered until software resets the Enable Count bit. When enabled, each High-to-Low T_{IN} transition causes T1 to reload and restart counting. Interrupt request IRQ5 is generated on every end-of-count.

6.6 CASCADING COUNTER/TIMERS

For some applications, it may be necessary to measure a time interval greater than a single counter/timer can measure. In this case, T_{IN} and T_{OUT} can be used to cascade T0 and T1 as a single unit (Figure 6-18). T0 should be configured to operate in Continuous mode and to drive T_{OUT}. T_{IN} should be configured as an external clock input to T1 and wired back to T_{OUT}. On every other T0 end-of-count, T_{OUT} undergoesaHigh-to-Lowtransitionthatcauses T1 tocount.

T1 can operate in either Single-Pass or Continuous mode. When the T1 end-of-count is reached, interrupt request IRQ5 is generated. Interrupt requests IRQ2 (T_{IN} High-to-Low transitions) and IRQ4 (T0 end-of-count) are also generated but are most likely of no importance in this configuration and should be disabled.

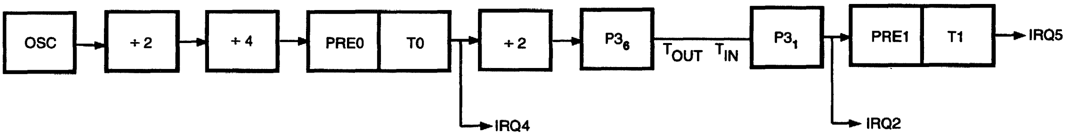


Figure 6-18. Cascaded Counter / Timers

6.7 RESET CONDITIONS

After a hardware reset, the counter/timers are disabled and the contents of the counter/timer and prescaler registers are undefined. However, the counting modes are configured for Single-Pass and the T1 clock source is set for external. T_{IN} is set for External Clock mode, and the T_{OUT} mode is off. Figures 6-19 through 6-22 show the binary reset values of the Prescaler, Counter/Timer, and Timer Mode registers.

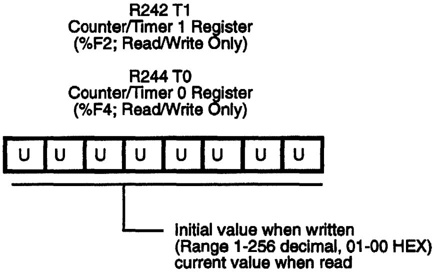


Figure 6-19. Counter / Timer Reset

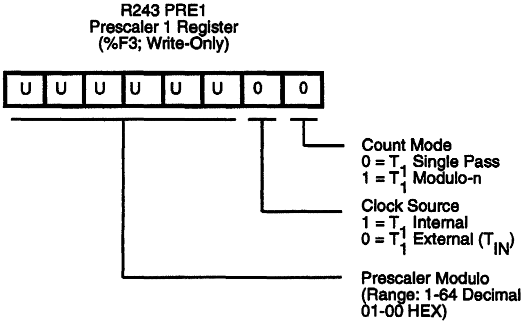


Figure 6-20. Prescaler 1 Register Reset

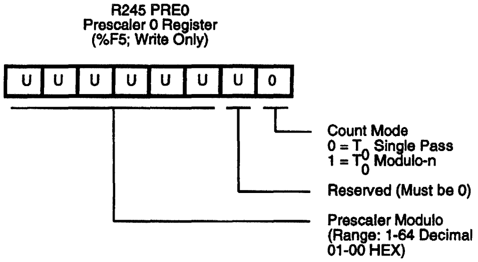


Figure 6-21. Prescaler 0 Reset

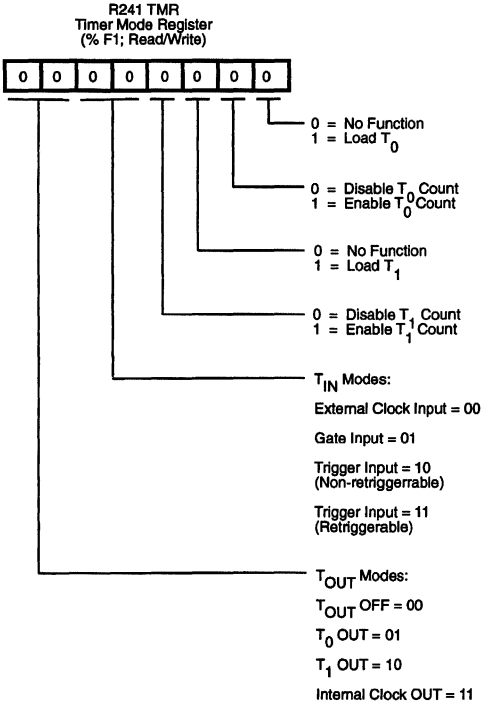


Figure 6-22. Timer ModeRßegister Reset

CHAPTER 7
INTERRUPTS

7.1 INTRODUCTION

The Z8® microcontroller allows six different interrupt levels from a variety of sources; up to four external inputs, the on-chip Counter/Timer(s), software, and serial I/O peripherals. These interrupts can be masked and their priorities set by using the Interrupt Mask and the Interrupt Priority Registers. All six interrupts can be globally disabled by resetting the master Interrupt Enable, bit 7 in the Interrupt Mask Register, with a Disable Interrupt (DI) instruction. Interrupts are globally enabled by setting bit 7 with an Enable Interrupt (EI) instruction.

There are three interrupt control registers: the Interrupt Request Register (IRQ), the Interrupt Mask register (IMR), and the Interrupt Priority Register (IPR). Figure 7-1 shows addresses and identifiers for the interrupt control registers. Figure 7-2 is a block diagram showing the Interrupt Mask and Interrupt Priority logic.

The Z8 MCU family supports both vectored and polled interrupt handling. Details on vectored and polled interrupts can be found later in this chapter.

Register	HEX	Identifier
Interrupt Mask	FBH	IMR
Interrupt Request	FAH	IRQ
Interrupt Priority	F9H	IPR

Figure 7-1. Interrupt Control Registers

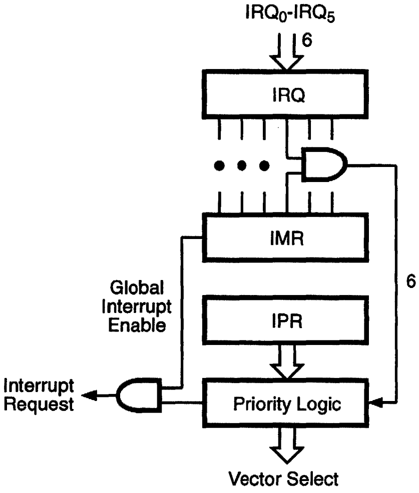


Figure 7-2. Interrupt Block Diagram

Note: See the selected Z8 MCU's product specification for the exact interrupt sources supported.

7.2 Interrupt Sources

Table 7-1 presents the interrupt types, sources, and vectors available in the Z8® family of processors.

Table 7-1. Interrupt Types, Sources, and Vectors *

Name	Sources	Vector Location	Comments
IRQ_0	$\overline{DAV}_0$, IRQ_0, Comparator	0,1	External ($P3_2$), Edge Triggered; Internal
IRQ_1	$\overline{DAV}_1$, IRQ_1	2,3	External ($P3_3$), Edge Triggered; Internal
IRQ_2	$\overline{DAV}_2$, IRQ_2, TIN, Comparator	4,5	External ($P3_1$), Edge Triggered; Internal
IRQ_3	IRQ_3	6,7	External ($P3_0$) or ($P3_2$), Edge Triggered; Internal
	Serial In	6,7	Internal
IRQ_4	T_0	8,9	Internal
	Serial Out	8,9	Internal
IRQ_5	T_1	10,11	Internal

7.2.1 External Interrupt Sources

External sources involve interrupt request lines IRQ0-IRQ3. IRQ0, IRQ1, and IRQ2 can be generated by a transition on the corresponding Port 3 pin (P32, P33, and P31 correspond to IRQ0, IRQ1, and IRQ2, respectively). Figure 7-3 is a block diagram for interrupt sources IRQ0, IRQ1, and IRQ2.

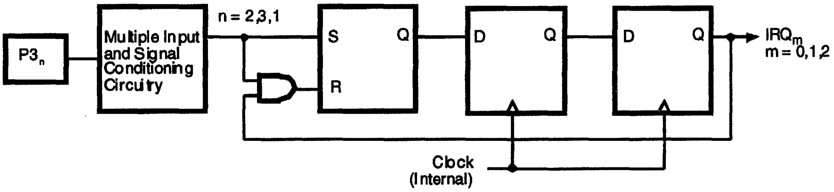


Figure 7-3. Interrupt Sources IRQ0-IRQ2 Block Diagram

Note: The interrupt sources and trigger conditions are device dependent. See the device product specification to determine available sources (internal and external), triggering edge options, and exact programming details.

When the Port 3 pin (P31, P32, or P33) transitions, the first flip-flop is set. The next two flip-flops synchronize the request to the internal clock and delay it by two internal clock periods. The output of the last flip-flop (IRQ0, IRQ1, or IRQ2) goes to the corresponding Interrupt Request Register.

IRQ3 can be generated from an external source only if Serial In is not enabled. Otherwise, its source is internal. The external request is generated by a negative edge signal on P30 as shown in Figure 7-4. Again, the external request is synchronized and delayed before reaching IRQ3. Some Z8® products replace P30 with P32 as the external source for IRQ3. In this case, IRQ3 interrupt generation follows the logic as illustrated in Figure 7-3.

Note: Although interrupts are edge triggered, minimum interrupt request Low and High times must be observed for proper operation. See the device product specification for exact timing requirements on external interrupt requests (T_wIL, T_wIH).

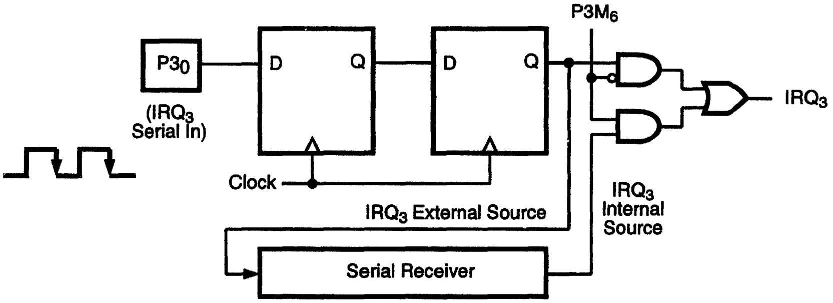


Figure 7-4. Interrupt Source IRQ3 Block Diagram

7.2.2 Internal Interrupt Sources

Internal sources involve interrupt requests IRQ0, IRQ1, IRQ3, IRQ4, and IRQ5. Internal sources are ORed with the external sources, so either an internal or external source can trigger the interrupt. Internal interrupt sources and trigger conditions are device dependent. See the device product specification to determine available sources, triggering edge options, and exact programming details.

For more details on the internal interrupt sources, refer to the chapters describing the Counter/Timer, I/O ports, and Serial I/O.

7.3 INTERRUPT REQUEST (IRQ) REGISTER LOGIC AND TIMING

Figure 7-5 shows the logic diagram for the Interrupt Request (IRQ) Register. The leading edge of the request will set the first flip-flop, that will remain set until interrupt requests are sampled.

Requests are sampled internally during the last clock cycle before an opcode fetch (Figure 7-6). External requests are sampled two internal clocks earlier, due to the synchronizing flip-flops shown in Figures 7-3 and 7-4.

At sample time the request is transferred to the second flip-flop in Figure 7-5, that drives the interrupt mask and priority logic. When an interrupt cycle occurs, this flip-flop will be reset only for the highest priority level that is enabled.

The user has direct access to the second flip-flop by reading and writing the IRQ Register. IRQ is read by specifying it as the source register of an instruction and written by specifying it as the destination register.

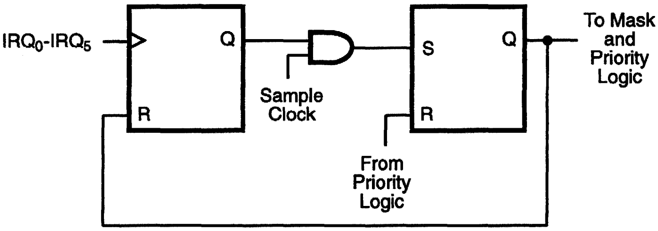


Figure 7-5. IRQ Register Logic

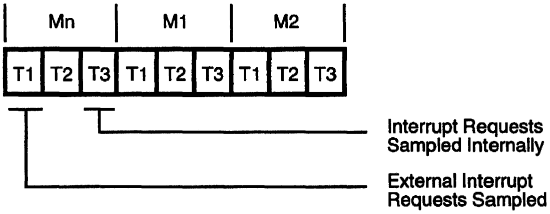

Mn
M1
M2
T1
T2
T3
T1
T2
T3
T1
T2
T3
Interrupt Requests Sampled Internally
External Interrupt Requests Sampled

Figure 7-6. Interrupt Request Timing

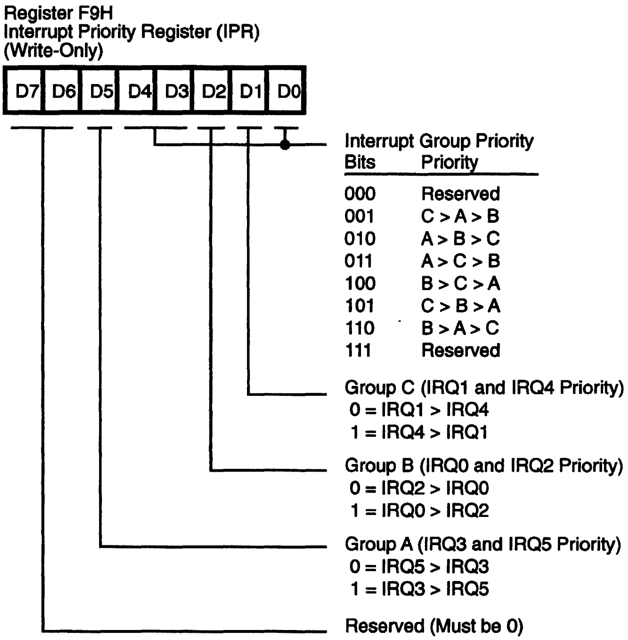
7.4 INTERRUPT INITIALIZATION

After reset, all interrupts are disabled and must be initialized before vectored or polled interrupt processing can begin. The Interrupt Priority Register (IPR), Interrupt Mask Register (IMR), and Interrupt Request Register (IRQ) must be initialized, in that order, to start the interrupt process. However, IPR need not be initialized for polled processing.

7.4.1 Interrupt Priority Register (IPR) Initialization

IPR (Figure 7-7) is a write-only register that sets priorities for the six levels of vectored interrupts in order to resolve simultaneous interrupt requests. (There are 48 sequence possibilities for interrupts.) The six interrupt levels IRQ0-IRQ5 are divided into three groups of two interrupt requests each. One group contains IRQ3 and IRQ5. The second group contains IRQ0 and IRQ2, while the third group contains IRQ1 and IRQ4.

Priorities can be set both within and between groups as shown in Tables 7-2 and 7-3. Bits 1, 2, and 5 define the priority of the individual members within the three groups. Bits 0, 3, and 4 are encoded to define six priority orders between the three groups. Bits 6 and 7 are reserved.



Register F9H
Interrupt Priority Register (IPR)
(Write-Only)

Interrupt Group Priority

Bits	Priority
000	Reserved
001	C > A > B
010	A > B > C
011	A > C > B
100	B > C > A
101	C > B > A
110	B > A > C
111	Reserved

Group C (IRQ1 and IRQ4 Priority)
0 = IRQ1 > IRQ4
1 = IRQ4 > IRQ1

Group B (IRQ0 and IRQ2 Priority)
0 = IRQ2 > IRQ0
1 = IRQ0 > IRQ2

Group A (IRQ3 and IRQ5 Priority)
0 = IRQ5 > IRQ3
1 = IRQ3 > IRQ5

Reserved (Must be 0)

Figure 7-7. Interrupt Priority Register

Table 7-2. Interrupt Priority

			Priority	
Group	Bit	Value	Highest	Lowest
C	Bit 1	0	IRQ1	IRQ4
		1	IRQ4	IRQ1
B	Bit 2	0	IRQ2	IRQ0
		1	IRQ0	IRQ2
A	Bit 5	0	IRQ5	IRQ3
		1	IRQ3	IRQ5

Table 7-3. Interrupt Group Priority

Bit Pattern			Group Priority		
Bit 4	Bit 3	Bit 0	High	Medium	Low
0	0	0	Not Used		
0	0	1	C	A	B
0	1	0	A	B	C
0	1	1	A	C	B
1	0	0	B	C	A
1	0	1	C	B	A
1	1	0	B	A	C
1	1	1	Not Used		

7.4.2 Interrupt Mask Register (IMR) Initialization

MR individually or globally enables or disables the six interrupt requests (Figure 7-8). When bit 0 to bit 5 are set to 1, the corresponding interrupt requests are enabled. Bit 7 is the master enable and must be set before any of the individual interrupt requests can be recognized. Resetting bit 7 globally disables all the interrupt requests. Bit 7 is set and reset by the EI and DI instructions. It is automatically reset during an interrupt service routine and set following the execution of an Interrupt Return (IRET) instruction.

Note: Bit 7 must be reset by the DI instruction before the contents of the Interrupt Mask Register or the Interrupt Priority Register are changed except:

- Immediately after a hardware reset.
- Immediately after executing an interrupt service routine and before IMR bit 7 has been set by any instruction.

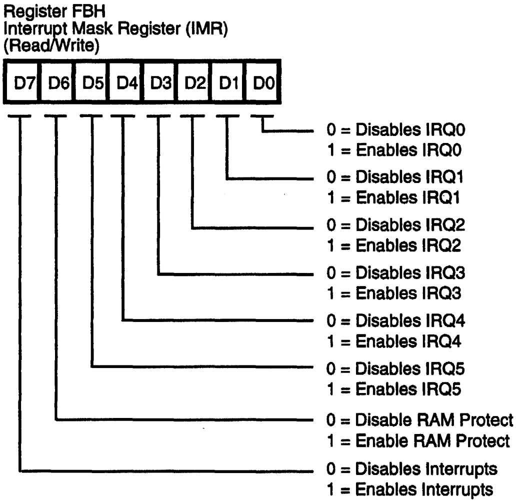


Figure 7-8. Interrupt Mask Register

Note: The RAM Protect option is selected at ROM mask submission time or at EPROM program time. If not selected or not an available option, this bit is reserved and must be 0.

7.4.3 Interrupt Request (IRQ) Register Initialization

IRQ (Figure 7-9) is a read/write register that stores the interrupt requests for both vectored and polled interrupts. When an interrupt is made on any of the six levels, the corresponding bit position in the register is set to 1. Bit 0 to bit 5 are assigned to interrupt requests IRQ0 to IRQ5, respectively.

Whenever Power-On Reset (POR) is executed, the IRQ resister is reset to 00H and disabled. Before the IRQ register will accept requests, it must be enabled by executing an ENABLE INTERRUPTS (EI) instruction.

Note: Setting the Global Interrupt Enable bit in the Interrupt Mask Register (IMR, bit 7) will not enable the IRQ. Execution of the EI instruction is required (Figure 7-10).

For polled processing, IRQ must still be initialized by an EI instruction.

To properly initialize the IRQ register, the following code is provided:

```
CLR    IMR
EI
DI
```

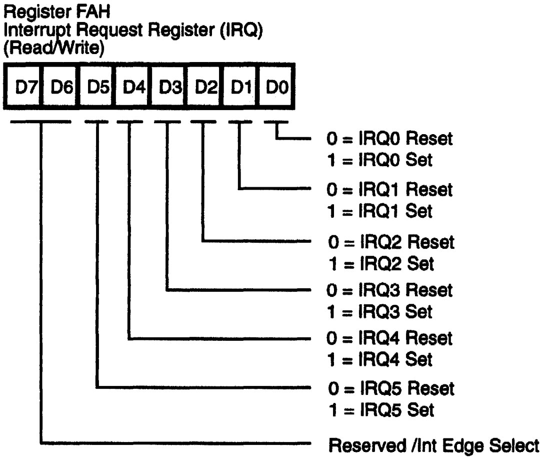


Figure 7-9. Interrupt Request Register

IMR is cleared before the IRQ enabling sequence to insure no unexpected interrupts occur when EI is executed. This code sequence should be executed prior to programming the application required values for IPR and IMR.

Note: IRQ bits 6 and 7 are device dependent. When reserved, the bits are not used and will return a 0 when read. When used as the Interrupt Edge select bits, the configuration options are as show in Table 7-4.

Table 7-4. IRQ Register Configuration

IRQ		Interrupt Edge	
D7	**D6**	**P31**	**P32**
0	0	F	F
0	1	F	R
1	0	R	F
1	1	R/F	R/F

Note:
F = Falling Edge
R = Rising Edge

The proper sequence for programming the interrupt edge select bits is (assumes IPR and IMR have been previously initialized):

```
DI                          ;Inhibit all interrupts
                            till input edges are
                            configured.
OR    IRQ,#XX 000000B       ;Configure interrupt
                            edges as needed -
                            do not disturb
                            IRQ 0-5.
EI                          ;Re-enable inter-
                            rupts.
```

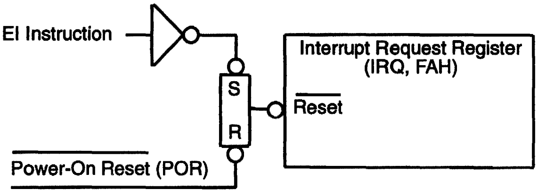

EI Instruction
S
R
$\overline{\text{Power-On Reset (POR)}}$
Interrupt Request Register (IRQ, FAH)
$\overline{\text{Reset}}$

Figure 7-10. IRQ Reset Functional Logic Diagram

7.5 IRQ SOFTWARE INTERRUPT GENERATION

IRQ can be used to generate software interrupts by specifying IRQ as the destination of any instruction referencing the Z8® Standard Register File. These Software Interrupts (SWI) are controlled in the same manner as hardware generated requests (in other words, the IPR and the IMR control the priority and enabling of each SWI level).

To generate a SWI, the desired request bit in the IRQ is set as follows:

```
        OR      IRQ, #NUMBER
```

where the immediate data, NUMBER, has a 1 in the bit position corresponding to the level of the SWI desired. For example, if an SWI is desired on IRQ5, NUMBER would have a 1 in bit 5:

```
        OR      IRQ, #00100000B
```

With this instruction, if the interrupt system is globally enabled, IRQ5 is enabled, and there are no higher priority pending requests, control is transferred to the service routine pointed to by the IRQ5 vector.

7.6 VECTORED PROCESSING

Each Z8 interrupt level has its own vector. When an interrupt occurs, control passes to the service routine pointed to by the interrupt's vector location in program memory. The sequence of events for vectored interrupts is as follows:

- PUSH PC Low Byte on Stack
- PUSH PC High Byte on Stack
- PUSH FLAGS on Stack
- Fetch High Byte of Vector
- Fetch Low Byte of Vector
- Branch to Service Routine specified by Vector

Figures 7-11 and 7-12 show the vectored interrupt operation.

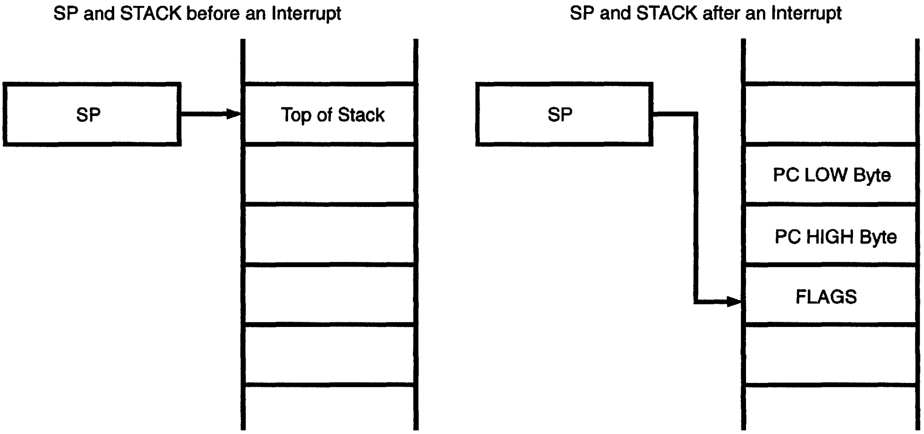

SP and STACK before an Interrupt
SP
Top of Stack
SP and STACK after an Interrupt
SP
PC LOW Byte
PC HIGH Byte
FLAGS

Figure 7-11. Effects of an Interrupt on the STACK

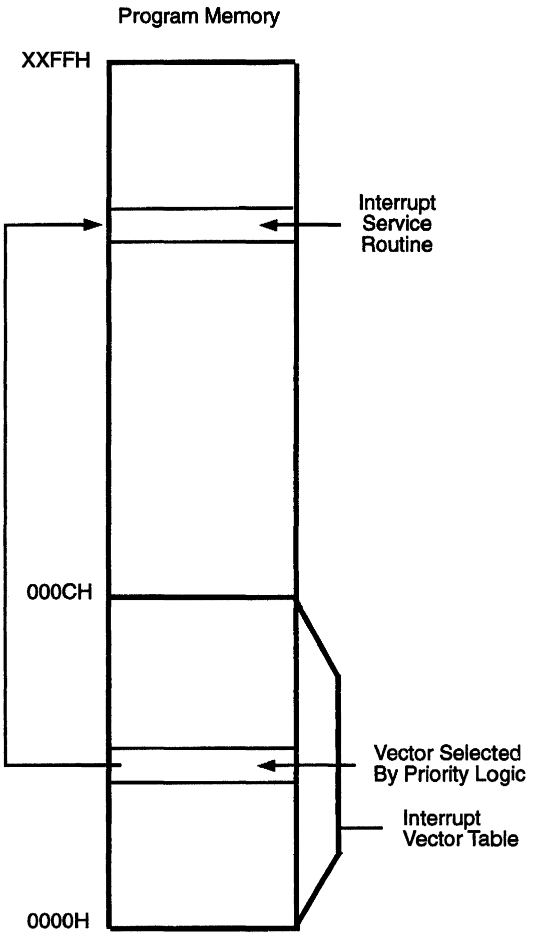


Figure 7-12. Interrupt Vectoring

7.6.1 Vectored Interrupt Cycle Timing

The interrupt acknowledge cycle time is 24 internal clock cycles and is shown in Figure 7-13. In addition, two internal clock cycles are required for the synchronizing flip-flops. The maximum interrupt recognition time is equal to the number of clock cycles required for the longest executing instruction present in the user program (assumes worst case condition of interrupt sampling, Figure 7-6 , just prior to the interrupt occurrence). To calculate the worst case interrupt latency (maximum time required from interrupt generation to fetch of the first instruction of the interrupt service routine), sum these components:

Worst Case Interrupt Latency ≈ 24 TpC (interrupt acknowledge time) + # T_PC of longest instruction present in the user's application program + $2T_PC$ (internal synchronization time).

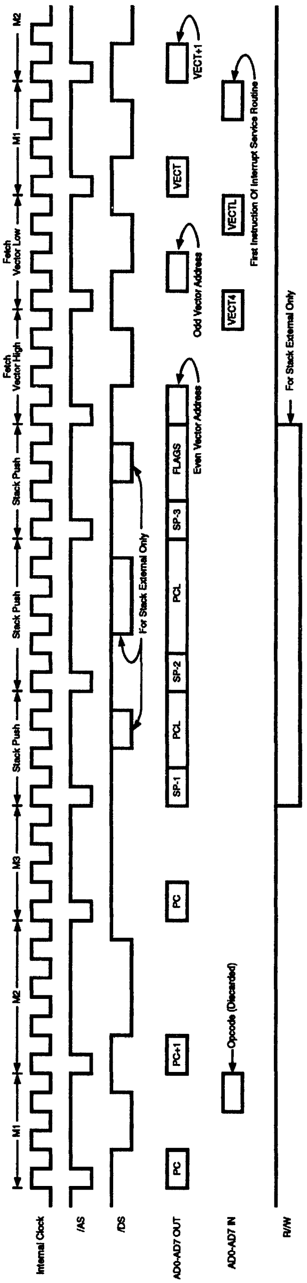


Figure 7-13. Z8 Interrupt Acknowledge Timing

7.6.2 Nesting of Vectored Interrupts

Nesting of vectored interrupts allows higher priority requests to interrupt a lower priority request. To initiate vectored interrupt nesting, do the following during the interrupt service routine:

- Push the old IMR on the stack.
- Load IMR with a new mask to disable lower priority interrupts.
- Execute EI instruction.
- Proceed with interrupt processing.
- After processing is complete, execute DI instruction.
- Restore the IMR to its original value by returning the previous mask from the stack.
- Execute IRET.

Depending on the application, some simplification of the above procedure may be possible.

7.7 POLLED PROCESSING

Polled interrupt processing is supported by masking off the IRQ levels to be polled. This is accomplished by clearing the corresponding bits in the IMR.

To initiate polled processing, check the bits of interest in the IRQ using the Test Under Mask (TM) instruction. If the bit is set, call or branch to the service routine. The service routine services the request, resets its Request Bit in the IRQ, and branches or returns back to the main program. An example of a polling routine is as follows:

```
        TM      IRQ, #MASKA    ;Test for request
        JR      Z, NEXT        ;If no request go to NEXT
        CALL    SERVICE        ;If request is there, then
                               ;service it
NEXT:
        .
        .
        .
SERVICE:                       ;Process Request
        .
        .
        .
        AND IRQ, #MASKB        ;Clear Request Bit
        RET                    ;Return to next
```

In this example, if IRQ2 is being polled, MASKA will be 00000100B and MASKB will be 11111011B.

7.8 RESET CONDITIONS

Upon reset, all bits in IPR are undefined.

In IMR, bit 7 is 0 and bits 0-6 are undefined. The IRQ register is reset and held in that state until an enable interrupt (EI) instruction is executed.

CHAPTER 8
POWER-DOWN MODES

8.1 INTRODUCTION

In addition to the standard RUN mode, the Z8® supports two Power-Down modes to minimize device current consumption. The two modes supported are HALT and STOP.

8.2 HALT MODE OPERATION

The HALT mode suspends instruction execution and turns off the internal CPU clock. The on-chip oscillator circuit remains active so the internal clock continues to run and is applied to the Counter/Timer(s) and interrupt logic.

To enter the HALT mode, it is necessary to first flush the instruction pipeline to avoid suspending execution in mid-instruction. To do this, the application program must execute a NOP instruction (opcode = FFH) immediately before the HALT instruction (opcode 7FH), that is,

```
FF     NOP    ;clear the instruction pipeline
7F     HALT   ;enter HALT mode
```

The HALT mode is exited by interrupts, either externally or internally generated. Upon completion of the interrupt service routine, the user program continues from the instruction after HALT.

The HALT mode may also be exited via a POR/RESET activation or a Watch-Dog Timer (WDT) timeout. (See the product data sheet for WDT availability). In this case, program execution will restart at the reset restart address 000CH.

To further reduce power consumption in the HALT mode, some Z8 family devices allow dynamic internal clock scaling. Clock scaling may be accomplished on the fly by reprogramming bit 0 and/or bit1 of the STOP-Mode Recovery register (SMR). See Figure 8-1.

Note: Internal clock scaling directly effects Counter/Timer operation — adjustment of the prescaler and downcounter values may be required. To determine the actual HALT mode current (I_{CC1}) value for the various optional modes available, see the selected Z8® device's product specification.

8.3 STOP MODE OPERATION

The STOP mode provides the lowest possible device standby current. This instruction turns off the on-chip oscillator and internal system clock.

To enter the STOP mode, it is necessary to first flush the instruction pipeline to avoid suspending execution in mid-instruction. To do this, the application program must execute a NOP instruction (opcode=FFH) immediately before the STOP instruction (opcode=6FH), that is,

```
FF    NOP   ;clear the instruction pipeline
6F    STOP  ;enter STOP mode
```

The STOP mode is exited by any one of the following resets: Power-On Reset activation, WDT time out (if available), or a STOP-Mode Recovery source. Upon reset generation, the processor will always restart the application program at address 000CH.

POR/RESET activation is present on all Z8 devices and is implemented as a reset pin and/or an on-chip power on reset circuit.

Some Z8 devices allow for the on-chip WDT to run in the STOP mode. If so activated, the WDT timeout will generate a reset some fixed time period after entering the STOP mode.

Note: STOP-Mode Recovery by the WDT will increase the STOP mode standby current (I_{CC2}). This is due to the WDT clock and divider circuitry that is now enabled and running to support this recovery mode. See the product data sheet for actual Icc2 values.

All Z8 devices provide some form of dedicated STOP-Mode Recovery (SMR) circuitry. Two SMR methods are implemented — a single fixed input pin or a flexible, programmable set of inputs. The selected Z8 device product specification should be reviewed to determine the SMR options available for use.

Note: For devices that support SPI, the slave mode compare feature also serves as a SMR source.

In the simple case, a low level applied to input pin P27 will trigger a SMR. To use this mode, pin P27 (I/O Port 2, bit 7) must be configured as an input before the STOP mode is entered. The low level on P27 must meet a minimum pulse width T_{WSM}. (See the product data sheet) to trigger the device reset mode). Some Z8 devices provide multiple SMR input sources. The desired SMR source is selected via the SMR Register.

Note: Use of specialized SMR modes (P2.7 input or SMR register based) or the WDT timeout (only when in the STOP mode) provide a unique reset operation. Some control registers are initialized differently for a SMR/WDT triggered POR than a standard reset operation. See the product specification (register file map) for exact details.

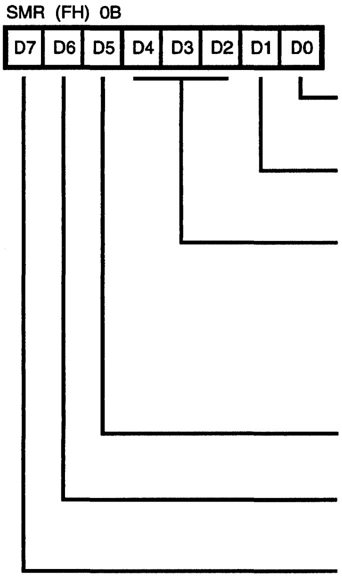
To determine the actual STOP mode current (I_{CC2}) value for the optional SMR modes available, see the selected Z8 device's product data sheet.

Note: The STOP mode current (I_{CC2}) will be minimized when:

- V_{CC} is at the low end of the devices operating range.
- WDT is off in the STOP mode.
- Output current sourcing is minimized.
- All inputs (digital and analog) are at the low or high rail voltages.

8.4 STOP-Mode Recovery Register (SMR)

This register selects the clock divide value and determines the mode of STOP-Mode Recovery (Figure 8-1). All bits are Write-Only, except bit 7, that is Read-Only. Bit 7 is a flag bit that is hardware set on the condition of STOP recovery and reset by a power-on cycle. Bit 6 controls whether a low level or a high level is required from the recovery source. Bit 5 controls the reset delay after recovery. Bits 2, 3, and 4, of the SMR register, specify the source of the STOP-Mode Recovery signal. Bits 0 and 1 control internal clock divider circuitry. The SMR is located in Bank F of the Expanded Register File at address 0BH.



SCLK/TCLK Divide-by-16
0 OFF * *
1 ON

External Clock Divide by 2
0 SCLK/TCLK =XTAL/2*
1 SCLK/TCLK =XTAL

STOP-Mode Recovery Source

000	POR Only and/or External Reset*
001	P30
010	P31
011	P32
100	P33
101	P27
110	P2 NOR 0-3
111	P2 NOR 0-7

Stop Delay
0 OFF
1 ON*

Stop Recovery Level
0 Low*
1 High

Stop Flag (Read only)
0 POR*
1 Stop Recovery

* Default setting after RESET.
* * Default setting after RESET and STOP-Mode Recovery.

Figure 8-1. STOP-Mode Recovery Register (Write-Only Except Bit D7, Which Is Read-Only)

Note: The SMR register is available in select Z8 MCU products. Refer to the device product specification to determine SMR options available.

SCLK/TCLK Divide-by-16 Select (DO). This bit of the SMR controls a divide-by-16 prescaler of SCLK/TCLK. The purpose of this control is to selectively reduce device power consumption during normal processor execution (SCLK control) and/or HALT mode (where TCLK sources counter/timers and interrupt logic).

External Clock Divide-by-Two (D1). This bit can eliminate the oscillator divide-by-two circuitry. When this bit is 0, the System Clock (SCLK) and Timer Clock (TCLK) are equal to the external clock frequency divided by two. The SCLK/TCLK is equal to the external clock frequency when this bit is set (D1=1). Using this bit together with D7 of PCON helps further lower EMI (D7 (PCON) =0, D1 (SMR) =1). The default setting is zero.

STOP-Mode Recovery Source (D2, D3, and D4). These three bits of the SMR specify the wake-up source of the STOP recovery and (Table 8-1 and Figure 8-2).

Table 8-1. STOP-Mode Recovery Source

SMR: 432			Operation
D4	D3	D2	Description of Action
0	0	0	POR and/or external reset recovery
0	0	1	P30 transition
0	1	0	P31 transition (not in Analog Mode)
0	1	1	P32 transition (not in Analog Mode)
1	0	0	P33 transition (not in Analog Mode)
1	0	1	P27 transition
1	1	0	Logical NOR of P20 through P23
1	1	1	Logical NOR of P20 through P27

STOP-Mode Recovery Delay Select (D5). This bit, if High, enables the T_{POR} /RESET delay after Stop-Mode Recovery. The default configuration of this bit is 1. If the "fast" wake up is selected, the Stop-Mode Recovery source is kept active for at least 5 TpC.

STOP-Mode Recovery Edge Select (D6). A 1 in this bit position indicates that a high level on any one of the recovery sources wakes the Z8® from STOP mode. A 0 indicates low-level recovery. The default is 0 on POR (Figure 8-2).

Cold or Warm Start (D7). This bit is set by the device upon entering STOP mode. A 0 in this bit (cold) indicates that the device reset by POR/WDT RESET. A 1 in this bit (warm) indicates that the device awakens by a SMR source.

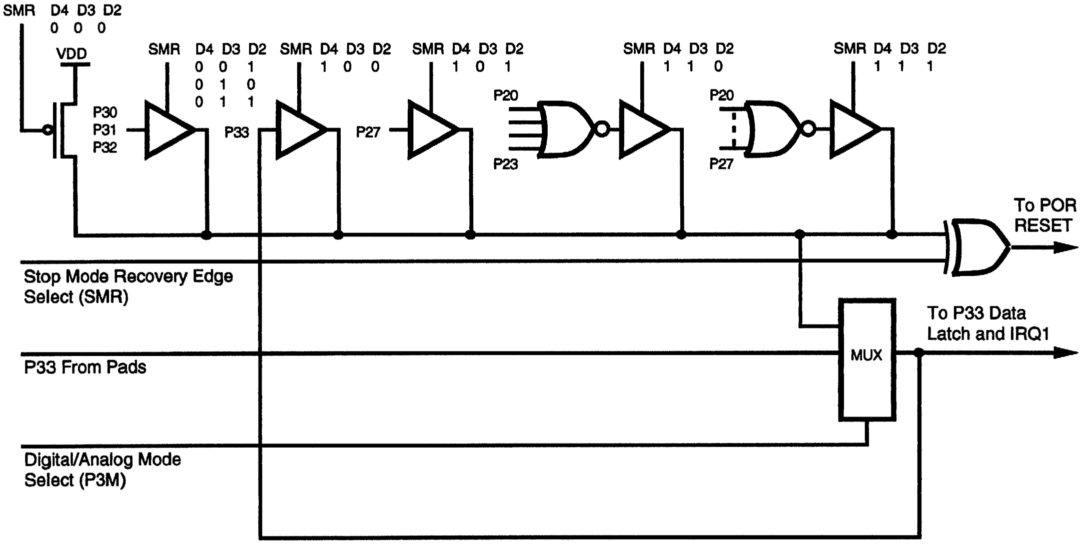


Figure 8-2. STOP-Mode Recovery Source

Note: If P31, P32, or P33 are to be used for a SMR source, the digital mode of operation must be selected prior to entering the STOP Mode.

CHAPTER 9
SERIAL I/O

9.1 UART INTRODUCTION

Select Z8® microcontrollers contain an on-board full-duplex Universal Asynchronous Receiver/Transmitter (UART) for data communications. The UART consists of a Serial I/O Register (SIO) located at address F0H, and its associated control logic (Figure 9-1). The SIO is actually two registers, the receiver buffer and the transmitter buffer, which are used in conjunction with Counter/Timer T0 and Port 3 I/O lines P30 (input) and P37 (output). Counter/Timer T0 provides the clock input for control of the data rates.

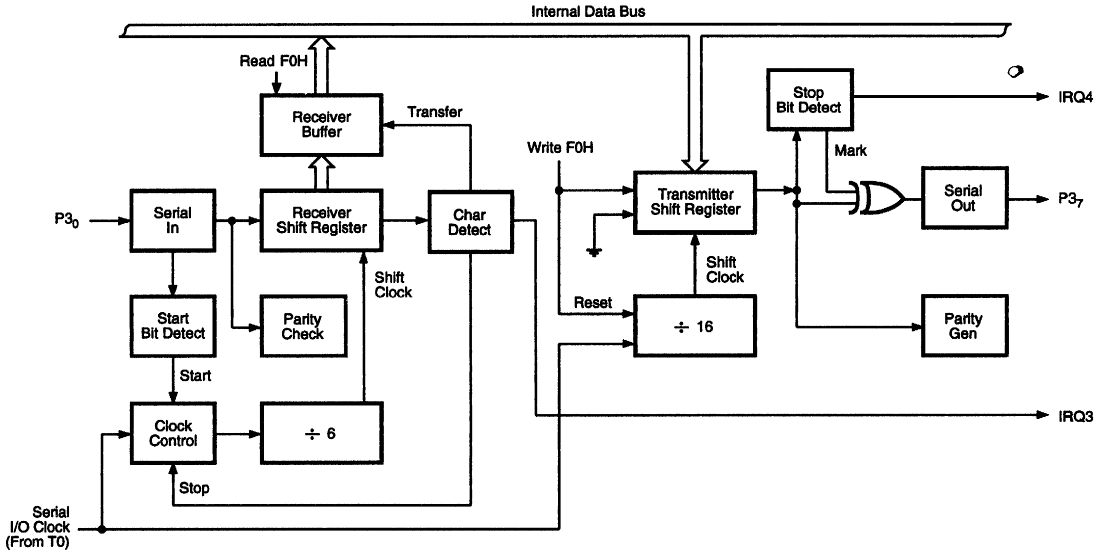


Figure 9-1. UART Block Diagram

Configuration of the UART is controlled by the Port 3 Mode Register (P3M) located at address F7H. The Z8® always transmits eight bits between the start and stop bits (eight Data Bits or seven Data Bits and one Parity Bit). Odd parity generation and detection is supported.

The SIO Register and its associated Mode Control Registers are mapped into the Standard Z8 Register File as shown in Table 9-1. The organization allows the software to access the UART as general-purpose registers, eliminating the need for special instructions.

Table 9-1. UART Register Map

Register Name	Identifier	Hex Address
Port 3 Mode	P3M	F7
T0 Prescaler	PRE0	F5
Timer/Counter0	T0	F4
Timer Mode	TMR	F1
UART	SIO	F0

9.2 UART BIT-RATE GENERATION

When Port 3 Mode Register bit 6 is set to 1, the UART is enabled and T0 automatically becomes the bit rate generator (Figure 9-2). The end-of-count signal of T0 no longer generates Interrupt Request IRQ4. Instead, the signal is used as the input to the divide-by-16 counters (one each for the receiver and the transmitter) that clock the data stream.

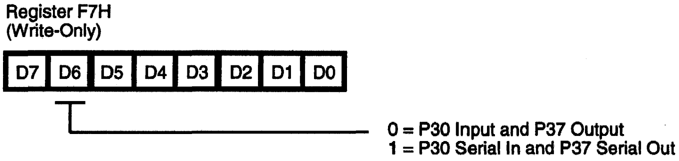

Register F7H
(Write-Only)
D7
D6
D5
D4
D3
D2
D1
D0
0 = P30 Input and P37 Output
1 = P30 Serial In and P37 Serial Out

Figure 9-2. Port 3 Mode Register (P3M) and Bit-Rate Generation

The divide chain that generates the bit rate is shown in Figure 9-3. The bit rate is given by the following equation:

Bit Rate = XTAL Frequency/(2 x 4 x p x t x 16)

where p and t are the initial values in Prescaler0 and Counter/Timer0, respectively. The final divide-by-16 is required since T0 runs at 16 times the bit rate in order to synchronize on the incoming data.

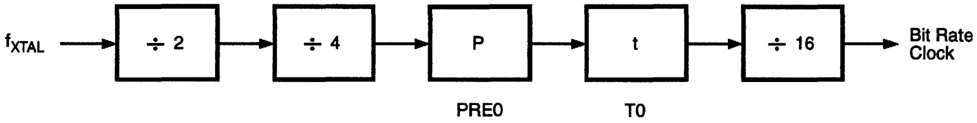


Figure 9-3. Bit Rate Divide Chain

To configure the Z8 for a specific bit rate, appropriate values as determined by the above equation must be loaded into registers PRE0 (F5H) and T0 (F4H). PRE0 also controls the counting mode for T0 and should therefore be set to the Continuous Mode (D0 = 1).

For example, given an input clock frequency (XTAL) of 11.9808 MHz and a selected bit rate of 1200 bits per second, the equation is satisfied by p = 39 and t = 2. Counter/Timer T0 should be set to 02H. With T0 in Continuous Mode, the value of PRE0 becomes 9DH (Figure 9-4).

Table 9-2 lists several commonly used bit rates and the values of XTAL, p, and t required to derive them. This list is presented for convenience and is not intended to be exhaustive.

Table 9-2. Bit Rates

Bit	7,3728		7,9872		9,8304		11,0592		11,6736		11,9808		12,2880	
Rate	**p**	**t**	**p**	**t**	**p**	**t**	**p**	**t**	**p**	**t**	**p**	**t**	**p**	**t**
19200	3	1	–	–	4	1	–	–	–	–	–	–	5	1
9600	3	2	–	–	4	2	9	1	–	–	–	–	5	2
4800	3	4	13	1	4	4	9	2	19	1	–	–	5	4
2400	3	8	13	2	4	8	9	4	19	2	39	1	5	8
1200	3	16	13	4	4	16	9	8	19	4	39	2	5	16
600	3	32	13	8	4	32	9	16	19	8	39	4	5	32
300	3	64	13	16	4	64	9	32	19	16	39	8	5	64
150	3	128	13	32	4	128	9	64	19	32	39	16	5	128
110	3	175	3	189	4	175	5	157	4	207	17	50	8	109

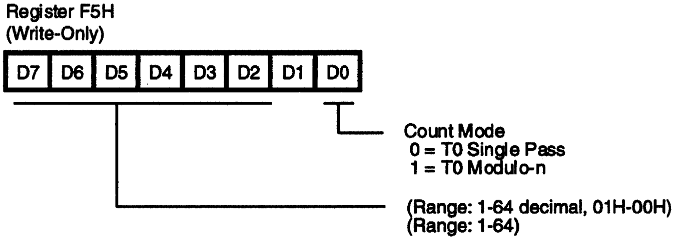


Figure 9-4. Prescaler 0 Register (PRE0) Bit-Rate Generation

The bit rate generator is started by setting the Timer Mode Register (TMR) (F1H) bit 1 and bit 0 both to 1 (Figure 9-5). This transfers the contents of the Prescaler 0 Register and Counter/Timer0 Register to their corresponding down counters. In addition, counting is enabled so that UART operations begin.

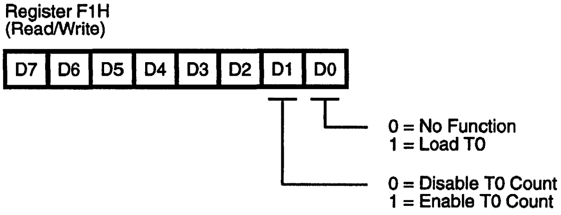


Figure 9-5. Timer Mode Register (TMR) Bit Rate Generation

9.3 UART RECEIVER OPERATION

The receiver consists of a receiver buffer (SIO Register [F0H]), a serial-in, parallel-out shift register, parity checking, and data synchronizing logic. The receiver block diagram is shown as part of Figure 9-1.

9.3.1 Receiver Shift Register

After a hardware reset or after a character has been received, the Receiver Shift Register is initialized to all 1s and the shift clock is stopped. Serial data, input through Port 3 bit 0, is synchronized to the internal clock by two D-type flip-flops before being input to the Shift Register and the start bit detection circuitry.

The start bit detection circuitry monitors the incoming data stream, looking for a start bit (a High-to-Low input transition). When a start bit is detected, the shift clock logic is enabled. The T0 input is divided-by-16 and, when the count equals eight, the divider outputs a shift clock. This clock shifts the start bit into the Receiver Shift Register at the center of the bit time. Before the shift actually occurs, the input is rechecked to ensure that the start bit is valid. If the detected start bit is false, the receiver is reset and the process of looking for a start bit is repeated. If the start bit is valid, the data is shifted into the Shift Register every sixteen counts until a full character is assembled (Figure 9-6).

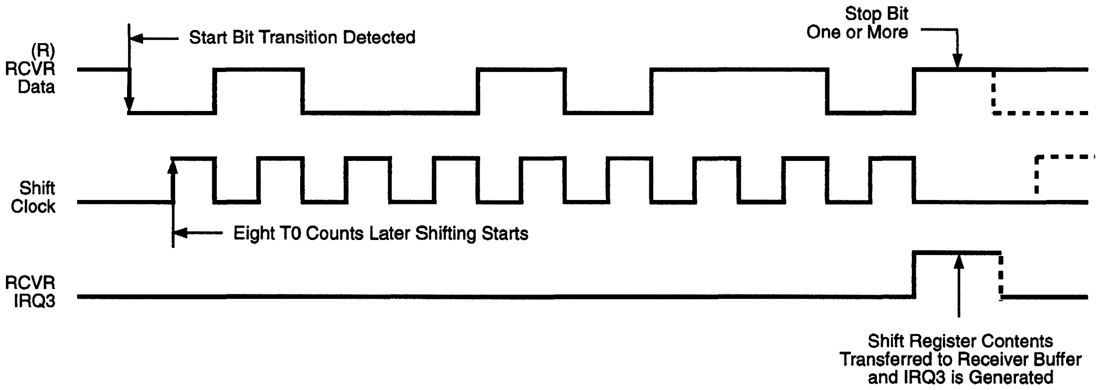

Start Bit Transition Detected
Stop Bit
One or More
(R)
RCVR
Data
Shift
Clock
Eight T0 Counts Later Shifting Starts
RCVR
IRQ3
Shift Register Contents
Transferred to Receiver Buffer
and IRQ3 is Generated

Figure 9-6. Receiver Timing

After a full character has been assembled in the receiver's buffer, SIO Register (F0H), Interrupt Request IRQ3 is generated. The shift clock is stopped and the Shift Register reset to all 1s. The start bit detection circuitry begins monitoring the data input for the next start bit. This cycle allows the receiver to synchronize on the center of the bit time for each incoming character.

9.3.2 Overwrites

Although the receiver is single buffered, it is not protected from being overwritten, so the software must read the SIO Register (F0H) within one character time after the interrupt request (IRQ3). The Z8 does not have a flag to indicate this overrun condition. If polling is used, the IRQ3 bit in the Interrupt Request Register must be reset by software.

9.3.3 Framing Errors

Framing error detection is not supported by the receiver hardware, but by responding to the interrupt request within one character bit time, the software can test for a stop bit on P30. Port 3 bits are always readable, which facilitates break detection. For example, if a null character is received, testing P30 results in a 0 being read.

9.3.4 Parity

The data format supported by the receiver must have a start bit, eight data bits, and at least one stop bit. If parity is on, bit 7 of the data received will be replaced by a Parity Error Flag. A parity error sets bit 7 to 1, otherwise, bit D7 is set to 0. Figure 9-7 shows these data formats.

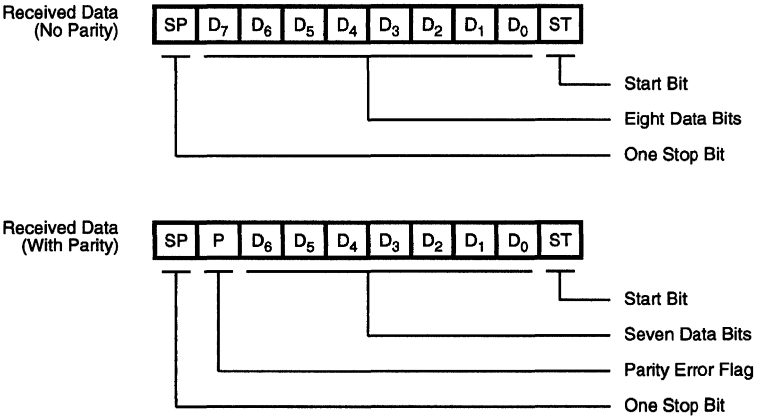


Figure 9-7. Receiver Data Formats

The Z8® hardware supports odd parity only, that is enabled by setting the Port 3 Mode Register bit 7 to 1 (Figure 9-8).

If even parity is required, the Parity Mode should be disabled (P3M bit 7 set to 0), and software must calculate the received data's parity.

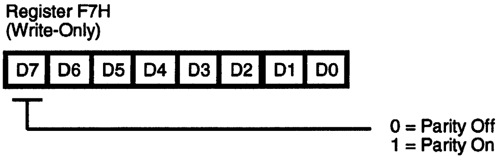


Figure 9-8. Port 3 Mode Register (P3M) Parity

9.4 TRANSMITTER OPERATION

The transmitter consists of a transmitter buffer (SIO Register [F0H]), a parity generator, and associated control logic. The transmitter block diagram is shown as part of Figure 9-1.

After a hardware reset or after a character has been transmitted, the transmitter is forced to a marking state (output always High) until a character is loaded into the transmitter buffer, SIO Register (F0H). The transmitter is loaded by specifying the SIO Register as the destination register of any instruction.

T0's output drives a divide-by-16 counter that in turn generates a shift clock every 16 counts. This counter is reset when the transmitter buffer is written by an instruction. This reset synchronizes the shift clock to the software. The transmitter then outputs one bit per shift clock, through Port 3 bit 7, until a start bit, the character written to the buffer, and two stop bits have been transmitted. After the second stop bit has been transmitted, the output is again forced to a marking state. Interrupt request IRQ4 is generated at this time to notify the processor that the transmitter is ready to accept another character.

9.4.1 Overwrites

The user is not protected from overwriting the transmitter, so it is up to the software to respond to IRQ4 appropriately. If polling is used, the IRQ4 bit in the Interrupt Request Register must be reset.

9.4.2 Parity

The data format supported by the transmitter has a start bit, eight data bits, and at least two stop bits. If parity is on, bit 7 of the data transmitted will be replaced by an odd parity bit. Figure 9-9 shows the transmitter data formats.

Parity is enabled by setting Port 3 Mode Register bit 7 to 1. If even parity is required, the parity mode should be disabled (P3M bit 7 reset to 0), and software must modify the data to include even parity.

Since the transmitter can be overwritten, the user is able to generate a break signal. This is done by writing null characters to the transmitter buffer (SIO Register [F0H]) at a rate that does not allow the stop bits to be output. Each time the SIO Register is loaded, the divide-by-16 counter is resynchronized and a new start bit is output followed by data.

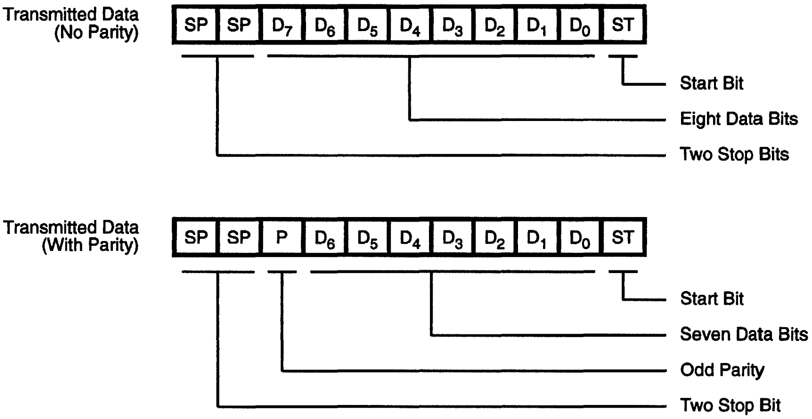


Figure 9-9. Transmitter Data Formats

9.5 UART RESET CONDITIONS

After a hardware reset, the SIO Register contents are undefined, and Serial Mode and parity are disabled. Figures 9-10 and 9-11 show the binary reset values of the SIO Register and its associated mode register P3M.

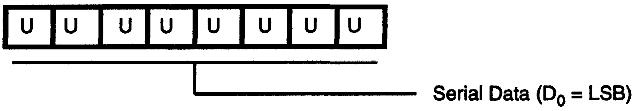


Figure 9-10. SIO Register Reset

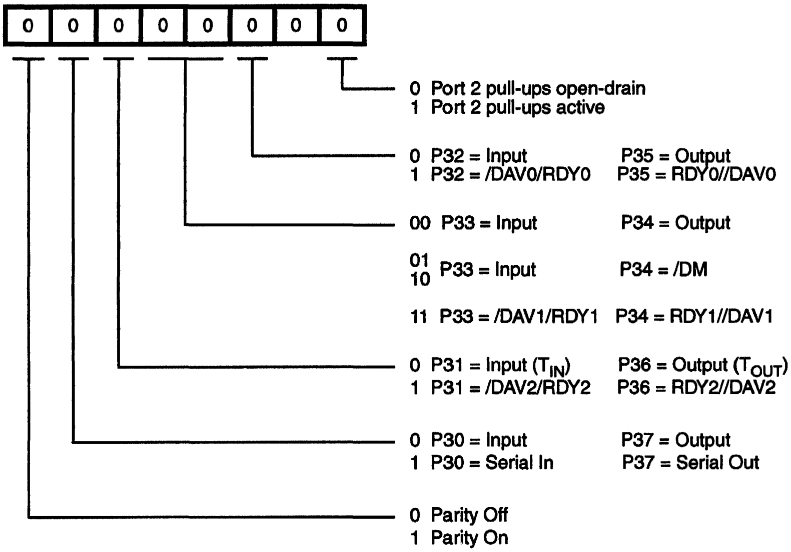

0 0 0 0 0 0 0 0
0 Port 2 pull-ups open-drain
1 Port 2 pull-ups active
0 P32 = Input P35 = Output
1 P32 = /DAV0/RDY0 P35 = RDY0//DAV0
00 P33 = Input P34 = Output
01
10 P33 = Input P34 = /DM
11 P33 = /DAV1/RDY1 P34 = RDY1//DAV1
0 P31 = Input (T_{IN}) P36 = Output (T_{OUT})
1 P31 = /DAV2/RDY2 P36 = RDY2//DAV2
0 P30 = Input P37 = Output
1 P30 = Serial In P37 = Serial Out
0 Parity Off
1 Parity On

Figure 9-11. P3M Register Reset

9.6 Serial Peripheral Interface (SPI)

Select Z8® microcontrollers incorporate a serial peripheral interface (SPI) for communication with other microcontrollers and peripherals. The SPI includes features such as Stop-Mode Recovery, Master/Slave selection, and Compare mode. Table 9-3 contains the pin configuration for the SPI feature when it is enabled. The SPI consists of four registers: SPI Control Register (SCON), SPI Compare Register (SCOMP), SPI Receive/Buffer Register (RxBUF), and SPI Shift Register. SCON is located in bank (C) of the Expanded Register File at address 02.

Table 9-3. SPI Pin Configuration

Name	Function	Pin Location
DI	Data-In	P20
DO	Data-Out	P27
SS	Slave Select	P35
SK	SPI Clock	P34

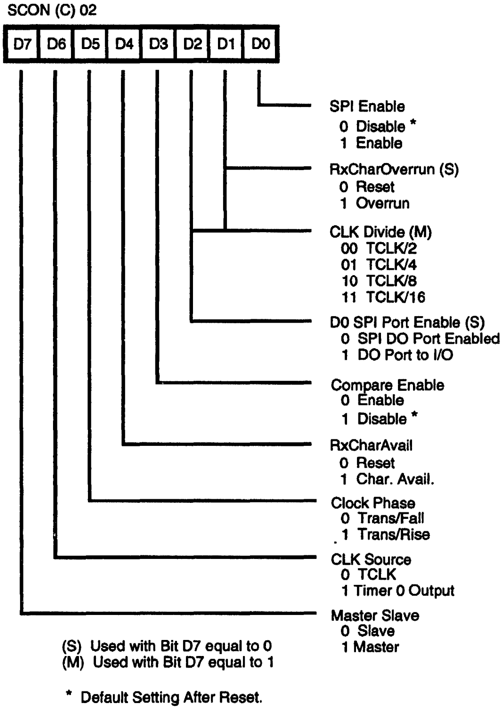


Figure 9-12. SPI Control Register (SCON)

The SPI Control Register (SCON) (Figure 9-12), is a read/write register that controls Master/Slave selection, interrupts, clock source and phase selection, and error flag. Bit 0 enables/disables the SPI with the default being SPI disabled. A 1 in this location will enable the SPI, and a 0 will disable the SPI. Bits 1 and 2 of the SCON register in Master Mode select the clock rate. The user may choose whether internal clock is divide-by-2, 4, 8, or 16. In Slave Mode, Bit 1 of this register flags the user if an overrun of the RxBUF Register has occurred. The RxCharOverrun flag is only reset by writing a 0 to this bit. In slave mode, bit 2 of the Control Register disables the data-out I/O function. If a 1 is written to this bit, the data-out pin is released to its original port configuration. If a 0 is written to this bit, the SPI shifts out one bit for each bit received. Bit 3 of the SCON Register enables the compare feature of the SPI, with the default being disabled. When the compare feature is enabled, a comparison of the value in the SCOMP Register is made with the value in the RxBUF Register. Bit 4 signals that a receive character is available in the RxBUF Register. If the associated IRQ3 is enabled, an interrupt is generated. Bit 5 controls the clock phase of the SPI. A 1 in bit 5 allows for receiving data on the clock's falling edge and transmitting data on the clock's rising edge. A 0 allows receiving data on the clock's rising edge and transmitting on the clock's falling edge. The SPI clock source is defined in bit 6. A 1 uses Timer0 output for the SPI clock, and a 0 uses TCLK for clocking the SPI. Finally, bit 7 determines whether the SPI is used as a Master or a Slave. A 1 puts the SPI into Master mode and a 0 puts the SPI into Slave mode.

9.7 SPI Operation

The SPI is used in one of two modes: either as system slave, or as system master. Several of the possible system configurations are shown in Figure 9-13. In the slave mode, data transfer starts when the slave select (SS) pin goes active. Data is transferred into the slave's SPI Shift Register through the DI pin, which has the same address as the RxBUF Register. After a byte of data has been received by the SPI Shift Register, a Receive Character Available (RCA/IRQ3) flag and interrupt is generated. The next byte of data will be received at this time. The RxBUF Register must be cleared, or a Receive Character Overrun (RxCharOverrun) flag will be set in the SCON Register, and the data in the RxBUF Register will be overwritten. When the communication between the master and slave is complete, the SS goes inactive.

Unless disconnected, for every bit that is transferred into the slave through the DI pin, a bit is transferred out through the DO pin on the opposite clock edge. During slave operation, the SPI clock pin (SK) is an input. In master mode, the CPU must first activate a SS through one of its I/O ports. Next, data is transferred through the master's DO pin one bit per master clock cycle. Loading data into the shift register initiates the transfer. In master mode, the master's clock will drive the slave's clock. At the conclusion of a transfer, a Receive Character Available (RCA/IRQ3) flag and interrupt is generated. Before data is transferred via the DO pin, the SPI Enable bit in the SCON Register must be enabled.

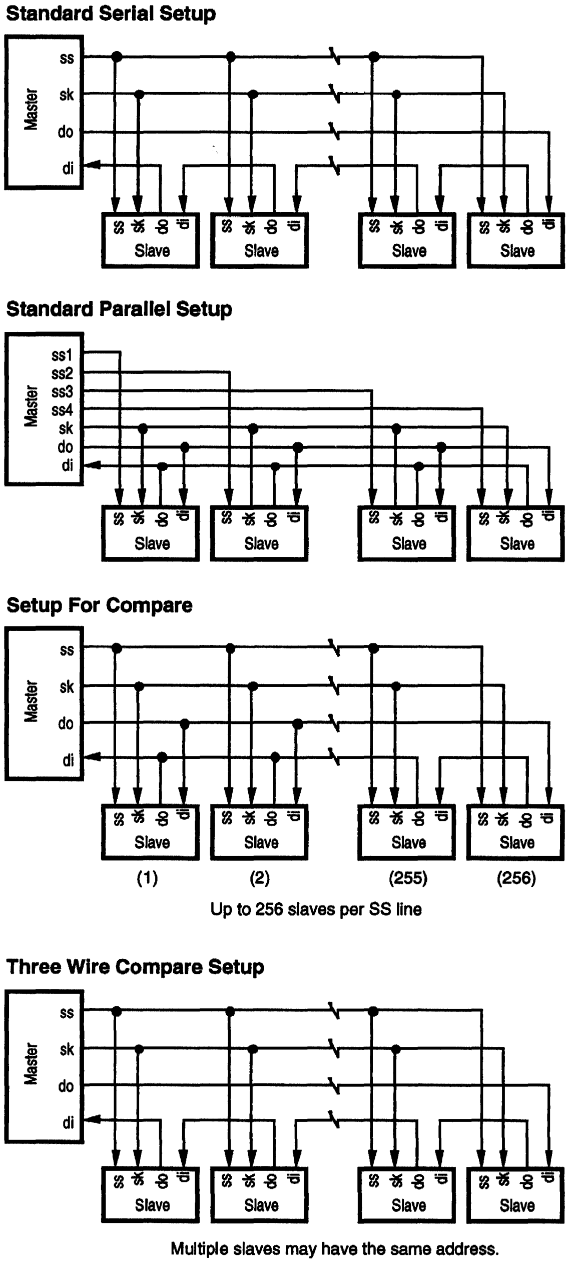
9.8 SPI Compare

When the SPI Compare Enable bit, D3 of the SCON Register is set to 1, the SPI Compare feature is enabled. The compare feature is only valid for slave mode. A compare transaction begins when the (SS) line goes active. Data is received as if it were a normal transaction, but there is no data transmitted to avoid bus contention with other slave devices. When the compare byte is received, IRQ3 is not generated. Instead, the data is compared with the contents of the SCOMP Register. If the data does not match, DO remains inactive and the slave ignores all data until the (SS) signal is reset. If the data received matches the data in the SCOMP register, then a SMR signal is generated. DO is activated if it is not tri-stated by D2 in the SCON Register, and data is received the same as any other SPI slave transaction.

When the SPI is activated as a slave, it operates in all system modes: STOP, HALT, and RUN. Slaves' not comparing remain in their current mode, whereas slaves' comparing wake from a STOP or HALT mode by means of an SMR.

9.9 SPI Clock

The SPI clock maybe driven by three sources: Timer0, a division of the internal system clock, or the external master when in slave mode. Bit D6 of the SCON Register controls what source drives the SPI clock. A 0 in bit D6 of the SCON Register determines the division of the internal system clock if this is used as the SPI clock source. Divide by 2, 4, 8, or 16 is chosen as the scaler.



Standard Serial Setup

Standard Parallel Setup

Setup For Compare

Up to 256 slaves per SS line

Three Wire Compare Setup

Multiple slaves may have the same address.

Figure 9-13. SPI System Configuration

9.10 Receive Character Available and Overrun

When a complete data stream is received, an interrupt is generated and the RxCharAvail bit in the SCON Register is set. Bit 4 in the SCON Register is for enabling or disabling the RxCharAvail interrupt. The RxCharAvail bit is available for interrupt polling purposes and is reset when the RxBUF Register is read. RxCharAvail is generated in both master and slave modes. While in slave mode, if the RxBUF is not read before the next data stream is received and loaded into the RxBUF Register, Receive Character Overrun (RxCharOverrun) occurs. Since there is no need for clock control in slave mode, bit D1 in the SPI Control Register is used to log any RxCharOverrun (Figure 9-14 and Figure 9-15).

No	Parameter	Min	Units
1	DI to SK Setup	10	ns
2	SK to D0 Valid	15	ns
3	SS to SK Setup	.5 Tsk	ns
4	SS to D0 Valid	15	ns
5	SK to DI Hold Time	10	ns

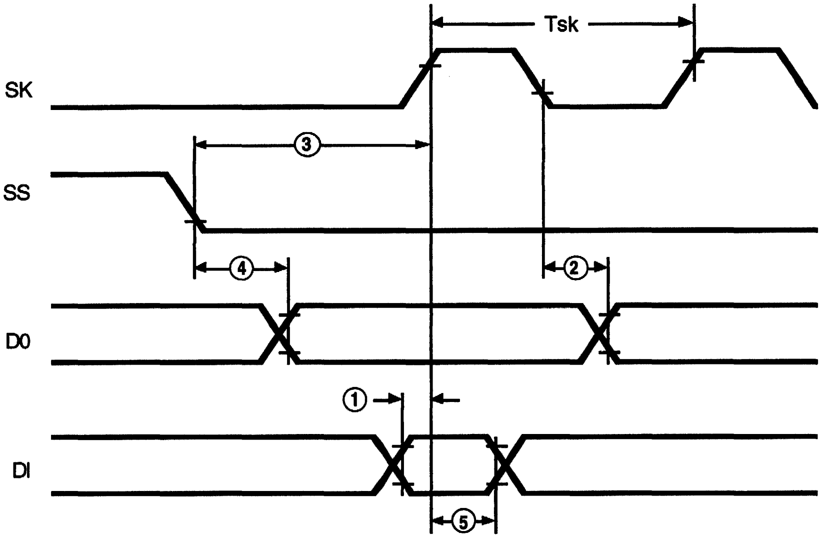


Figure 9-14. SPI Timing

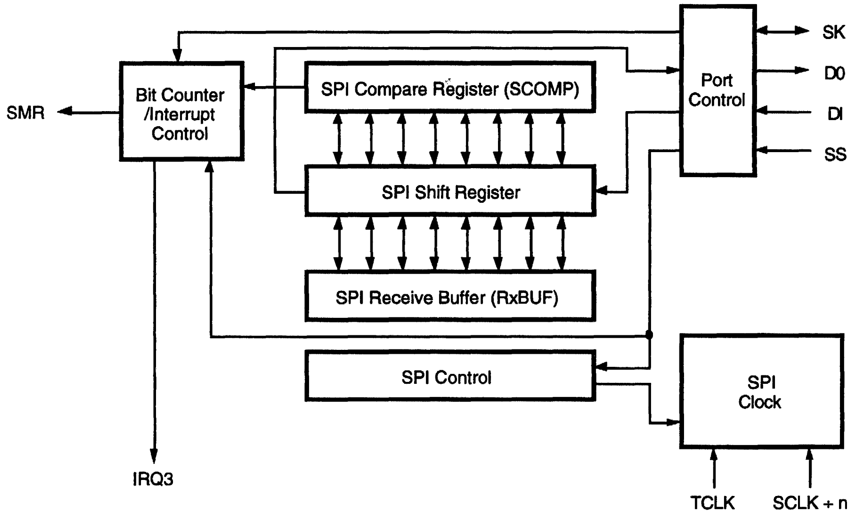

SK
D0
DI
SS
Port Control
Bit Counter /Interrupt Control
SMR
SPI Compare Register (SCOMP)
SPI Shift Register
SPI Receive Buffer (RxBUF)
SPI Control
SPI Clock
IRQ3
TCLK
SCLK ÷ n

Figure 9-15. SPI Logic

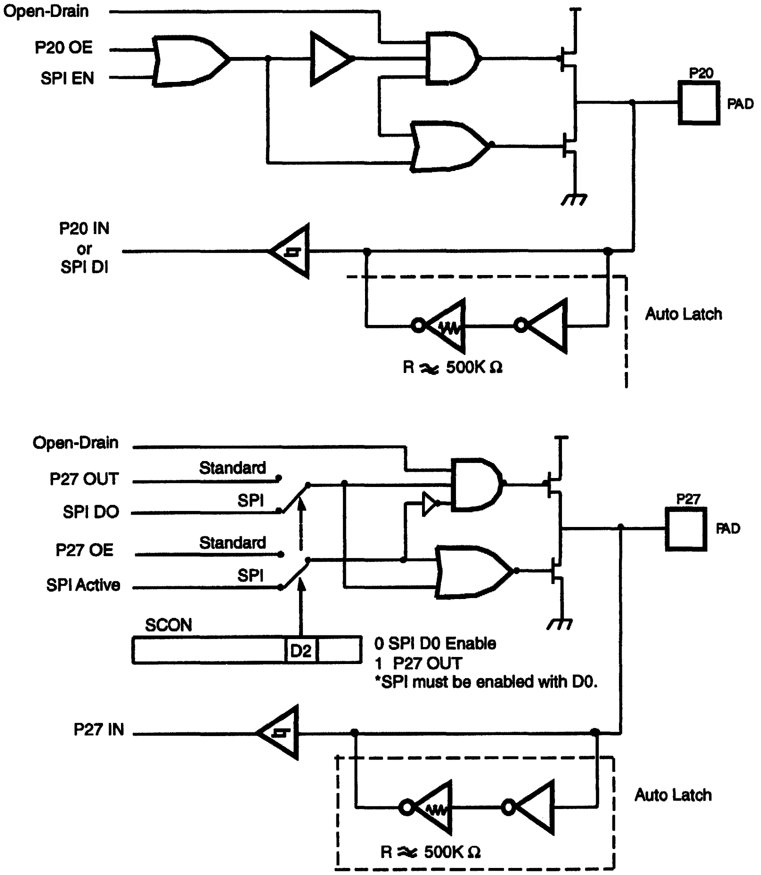


Figure 9-16. SPI Data In/Out Configuration

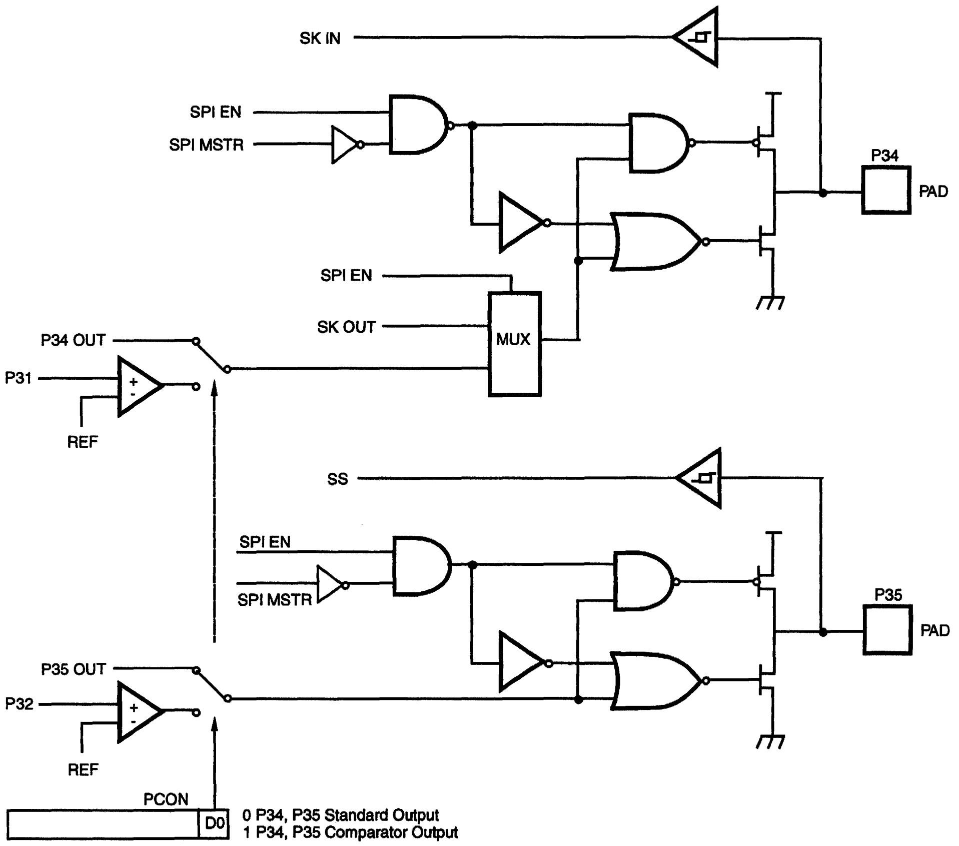


Figure 9-17. SPI Clock / SPI Slave Select Output Configuration

CHAPTER 10
EXTERNAL INTERFACE

10.1 INTRODUCTION

The Z8® can be a microcontroller with 20 pins for external memory interfacing. The external memory interface on the Z8 is generally for either RAM or ROM. This is only available for devices featuring Port 0, Port 1, R//W, /DM, /AS, and /DS. Please refer to specific product specifications for availability of these features.

The Z8 has a multiplexed external memory interface. In the multiplexed mode, eight pins from Port 1 form an Address/ Data Bus (AD7-AD0), eight pins from Port 0 form a High Address Bus (A15-A8). Three additional pins provide the Address Strobe, Data Strobe, and the Read/Write Signal. Figure 10-1 shows the external interface pins of the Z8.

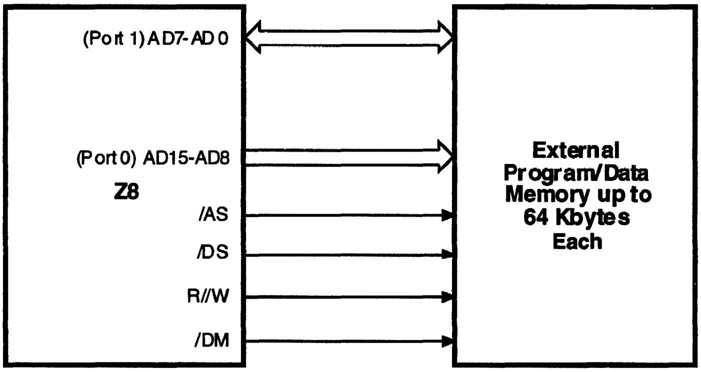


Figure 10-1. Z8 External Interface Pins

10.2 PIN DESCRIPTIONS

The following sections briefly describe the pins associated with the Z8® external memory interface.

10.2.1 /AS *Address Strobe* (output, active Low). Address Strobe is pulsed Low once at the beginning of each machine cycle. The rising edge of /AS indicates the address, Read/Write (R//W), and Data Memory (/DM) signals are valid for program or data memory transfers. In some cases, the Z8 address strobe is pulsed low regardless of accessing external or internal memory. Please refer to specific product specifications for /AS operation.

10.2.2 /DS *Data Strobe* (output, active Low). Data Strobe provides the timing for data movement to or from the Address/Data bus for each external memory transfer. During a Write Cycle, data out is valid at the leading edge of the /DS. During a Read Cycle, data in must be valid prior to the trailing edge of the /DS.

10.2.3 R//W *Read/Write* (output). Read/Write determines the direction of data transfer for memory transactions. R//W is Low when writing to program or data memory, and High for all other transactions.

10.2.4 /DM *Data Memory* (output). Data Memory provides a signal to separate External Program Memory from External Data Memory. It is a programmable function on pin P34. Data memory is active low for External Data Memory accesses and high for External Program Memory accesses.

10.2.5 P07 - P01 *High Address Lines* A15 -A8 (Outputs can be CMOS- or TTL- compatible. Please refer to product specifications for actual type). A15-A8 provide the High Address lines for the memory interface. Port 0 - 1 mode register must have bits D7 = 1 and D1 = 1 to configure Port 0 as A15 - A8 (Figure 10-2).

10.2.6 P17 - P10 *Address/Data Lines* AD7 - AD0 (inputs/outputs, TTL-compatible). AD7-AD0 is a multiplexed Address/Data memory interface. The lower eight Address lines (A7-A0) are multiplexed with Data lines (D7-D0). Port 0 - 1 mode register must have bits D4 = 1 and D3 = 0 to configure Port 1 as AD7 - AD0 (Figure 10-2).

10.2.7 /RESET *Reset* (input, active Low). /RESET initializes the Z8. When /RESET is deactivated, program execution begins from program location 000CH. If held Low, /RESET acts as a register file protect during power-down and power-up sequences. To avoid asynchronous and noisy reset problems, the Z8 is equipped with a reset filter of four external clocks ($4T_pC$). If the external /RESET signal is less than $4T_pC$ in duration, no reset will occur. On the fifth clock after the /RESET is detected, an internal reset signal is latched and held for an internal register count of 18 or more external clocks, or for the duration of the external /RESET, whichever is longer. Please refer to specific product specifications for length of reset delay time.

10.2.8 XTAL1, XTAL2. *Crystal1, Crystal2* (Oscillator input and output). These pins connect a parallel-resonant crystal, ceramic resonator, LC, RC network, or external single-phase clock to the on-chip oscillator input. Please refer to the device product specifications for information on availability of RC oscillator features.

10.3 EXTERNAL ADDRESSING CONFIGURATION

The minimum bus configuration uses Port 1 as a multiplexed address / data port (AD7 - AD0), allowing access to 256 bytes of external memory. In this configuration, the eight low order bits (A0 - A7) are multiplexed with the data (D7 - D0).

Port 0 can be programmed to provide either four additional address lines (A11 - A8), which increases the addressable memory to 4K bytes, or eight additional address lines (A15 - A8), which increases the addressable external memory up to 64K bytes. It is required to add a NOP after configuring Port 0 / Port 1 for external addressing before jumping to external memory execution.

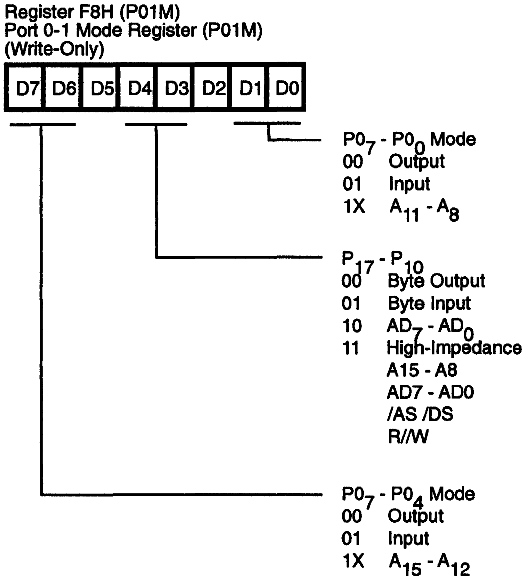


Figure 10-2. External Address Configuration

10.4 EXTERNAL STACKS

The Z8® architecture supports stack operations in either the Z8 Standard Register File or External Data Memory. A stack's location is determined by bit 2 in the Port 0-1 Mode Register (F8H). If bit 2 is set to 0, the stack is in External Data Memory. (Figure 10-3).

The instruction used to change the stack selection bit should not be immediately followed by the instructions RET or IRET, because this will cause indeterminate program flow. After a /RESET, the internal stack is selected.

Please note that if Port 0 is configured as A15 - A8 and the stack is selected as internal, any stack operation will cause the contents in register FEH to be displayed on Port 0.

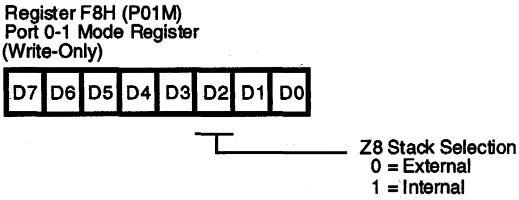
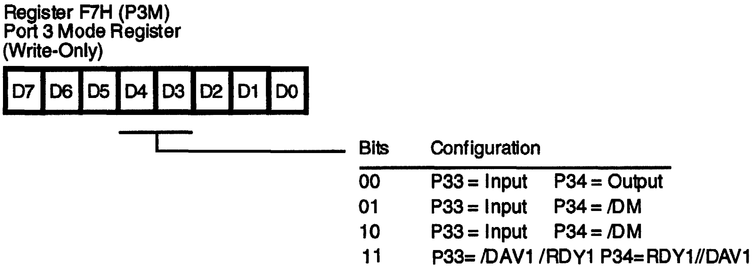


Figure 10-3. Z8 Stack Selection

10.5 DATA MEMORY

The two Z8 external memory spaces, data and program, are addressed as two separate spaces of up to 64 Kbytes each. External Program Memory and External Data Memory are logically selected by the Data Memory select output (/DM). /DM is made available on Port 3, bit 4 (P34) by setting bit 4 and bit 3 in the Port 3 Mode Register (F7H) to 10 or 01 (Figure 10-4). /DM is active Low during the execution of the LDE, LDEI instructions, and High for the execution of program instructions. /DM is also active Low during the execution of CALL, POP, PUSH, RET and IRET instructions if the stack resides in External Data Memory. After a /RESET, /DM is not selected.



Bits	Configuration
00	P33 = Input P34 = Output
01	P33 = Input P34 = /DM
10	P33 = Input P34 = /DM
11	P33= /DAV1 /RDY1 P34=RDY1//DAV1

Figure 10-4. Port 3 Data Memory Operation

10.6 BUS OPERATION

Typical data transfers between the Z8® and External Memory are illustrated in Figures 10-5 and 10-6. Machine cycles can vary from six to 12 clock periods depending on the operation being performed. The notations used to describe the basic timing periods of the Z8 are machine cycles (Mn), timing states (Tn), and clock periods. All timing references are made with respect to the output signals /AS and /DS. The clock is shown for clarity only and does not have a specific timing relationship with other signals.

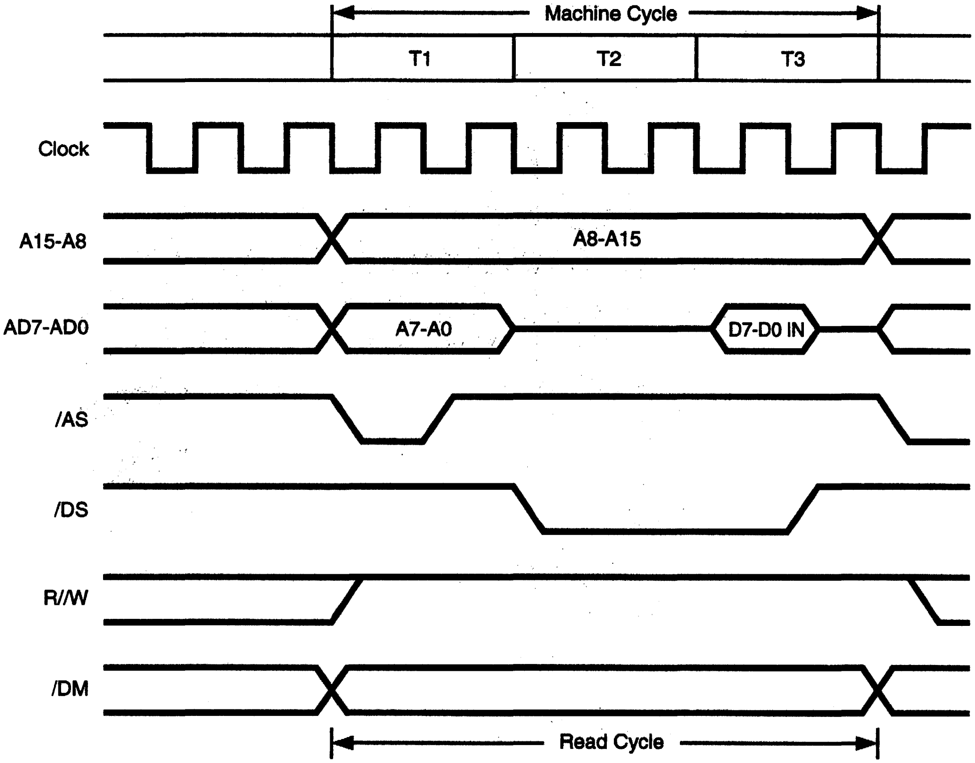


Figure 10-5. External Instruction Fetch or Memory Read Cycle

* Port inputs are strobed during T2, which is two internal system clocks before the execution cycle of the current instruction.

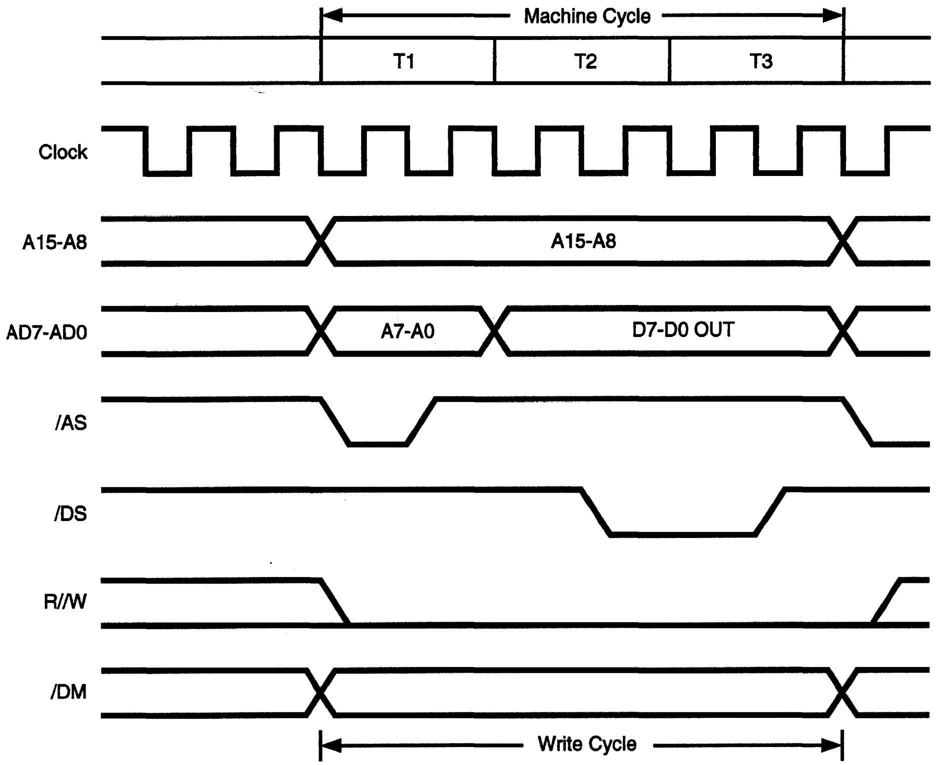


Figure 10-6. External Memory Write Cycle

10.6.1 Address Strobe (/AS)

All transactions start with /AS driven Low and then raised High by the Z8®. The rising edge of /AS indicates that R//W, /DM (if used), and the address outputs are valid. The address outputs (AD7-AD0), remain valid only during MnT1 and typically need to be latched using /AS. Address outputs (A15-A8) remain stable throughout the machine cycle, regardless of the addressing mode.

10.6.2 Data Strobe (/DS)

The Z8 uses /DS to time the actual data transfer. For Write operations (R//W = Low), a Low on /DS indicates that valid data is on the AD7-AD0 lines. For Read operations (R/W = High), the bus is placed in a high-impedance state before driving /DS Low, so the addressed device can put its data on the bus. The Z8 samples this data prior to raising /DS High.

10.7 EXTENDED BUS TIMING

Some products can accommodate slow memory access time by automatically inserting an additional software controlled state time (Tx). This stretches the /DS timing by two clock periods. Figures 10-7 and 10-8 illustrate extended external memory Read and Write cycles.

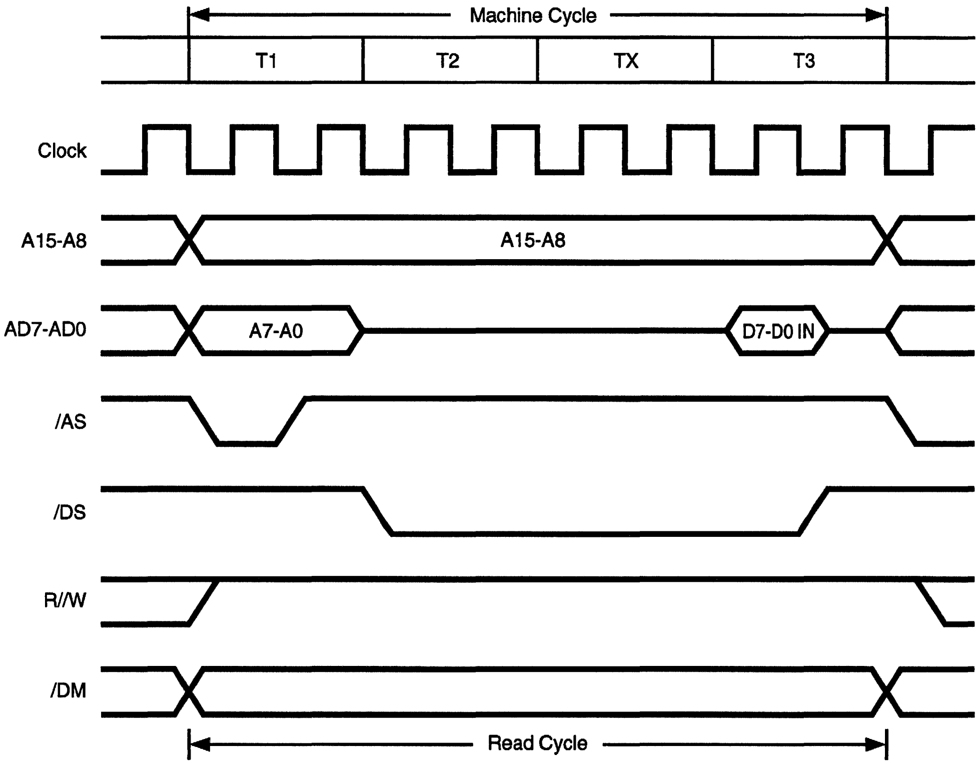


Figure 10-7. Extended External Instruction Fetch or Memory Read Cycle

* Port inputs are strobed during T2, which is two internal system clocks before the execution cycle of the current instruction.

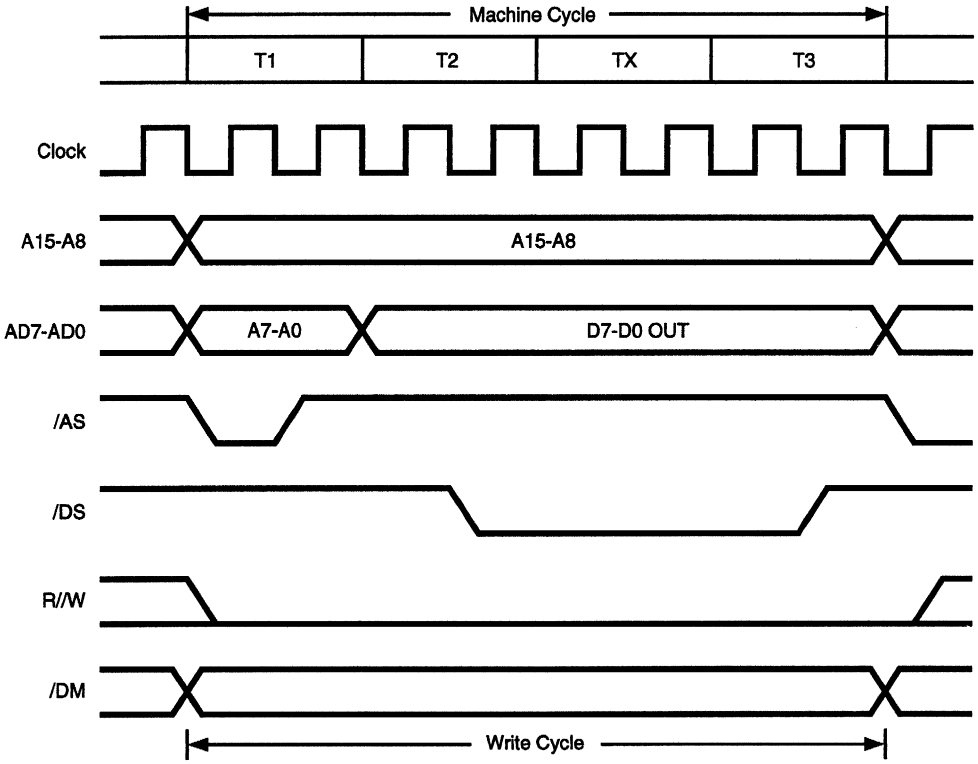


Figure 10-8. Extended External Memory Write Cycle

Timing is extended by setting bit D5 in the Port 0-1 Mode Register (F8H) to 1 (Figure 10-9). After a /RESET, this bit is set to 0.

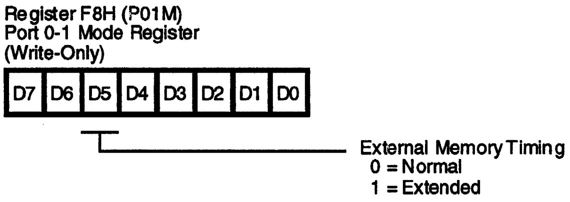


Figure 10-9. Extended Bus Timing

10.8 INSTRUCTION TIMING

The High throughput of the Z8® is due , in part, to the use of an instruction pipeline, in which the instruction fetch and execution cycles are overlapped. During the execution of the current instruction, the opcode of the next instruction is fetched. Instruction pipelining is illustrated in Figure 10-10.

Figures 10-11 and 10-12 show typical instruction cycle timing for instructions fetched from memory. For those instructions that require execution time longer than that of the overlapped fetch, or reference program or data memory as part of their execution, the pipe must be flushed.

Note: Figures 10-11 and 10-12 assume the XTAL/2 clock mode is selected.

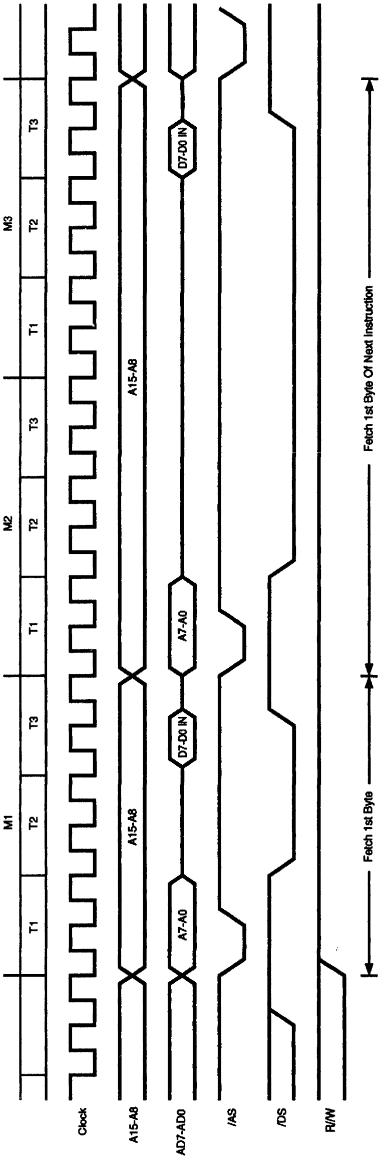


Figure 10-11. Instruction Cycle Timing (One-Byte Instructions)

* Port inputs are strobed during T2, which is two internal system clocks before the execution cycle of the current instruction.

* Port inputs are strobed during T2, which is two internal system clocks before the execution cycle of the current instruction.

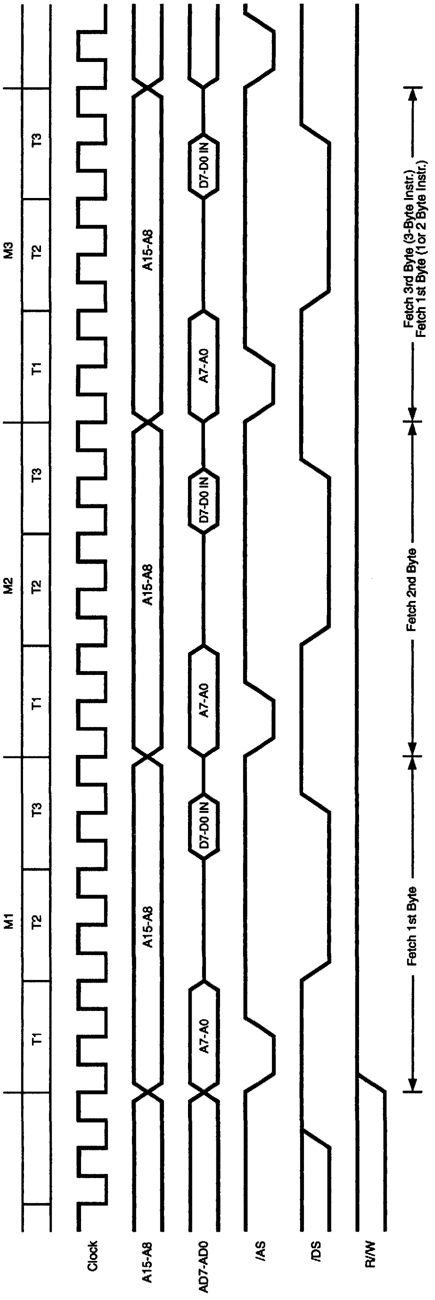


Figure 10-11. Instruction Cycle Timing (Two and Three Byte Instructions)

10.9 Z8 RESET CONDITIONS

After a hardware reset, extended timing is set to accommodate slow memory access during the configuration routine, /DM is inactive, the stack resides in the register file. Port 0, 1, and 2 are reset to input mode. Port 2 is set to Open-Drain Mode.

CHAPTER 11
ADDRESSING MODES

11.1 INTRODUCTION

11.1.1 Z8 Addressing Modes

The Z8® microcontroller provides six addressing modes:

- Register (R)
- Indirect Register (IR)
- Indexed (X)
- Direct (D)
- Relative (RA)
- Immediate (IM)

With the exception of immediate data and condition codes, all operands are expressed as register file, Program Memory, or Data Memory addresses. Registers are accessed using 8-bit addresses in the range of 00H-FFH. The Program Memory or Data Memory is accessed using 16-bit addresses (register pairs) in the range of 0000H-FFFFH.

Working Registers are accessed using 4-bit addresses in the range of 0-15 (0H-FH). The address of the register being accessed is formed by the combination of the upper four bits in the Register Pointer (R253) and the 4-bit working register address supplied by the instruction.

Registers can be used in pairs to designate 16-bit values or memory addresses. A Register Pair must be specified as an even-numbered address in the range of 0, 2, ..., 14 for Working Registers, or 4, 6, ...238 for actual registers.

In the following definitions of Z8 Addressing Modes, the use of 'register' can also imply register pair, working register, or working register pair, depending on the context.

Note: See the product data sheet for exact program, data, and register memory types and address ranges available.

11.2 Z8 REGISTER ADDRESSING (R)

In 8-bit Register Addressing mode, the operand value is equivalent to the contents of the specified register or register pair.

In the Register Addressing (Figure 11-1), the destination and/or source address specified corresponds to the actual register in the register file.

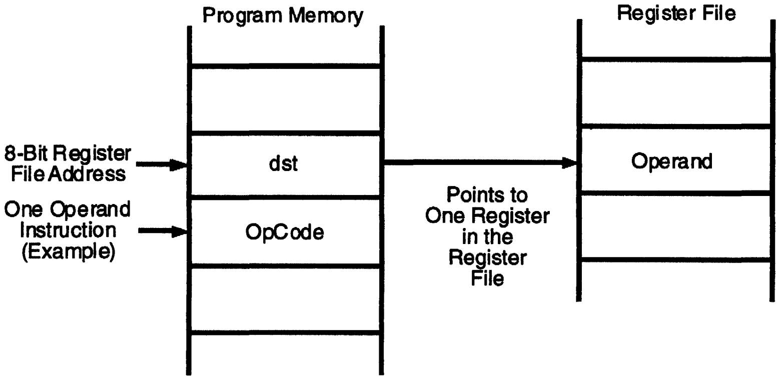

Program Memory
Register File
8-Bit Register File Address
One Operand Instruction (Example)
dst
OpCode
Points to One Register in the Register File
Operand

Figure 11-1. 8-Bit Register Addressing

In 4-bit Register Addressing (Figure 11-2), the destination and/or source addresses point to the Working Register within the current Working Register Group. This 4-bit address is combined with the upper four bits of the Register Pointer to form the actual 8-bit address of the affected register.

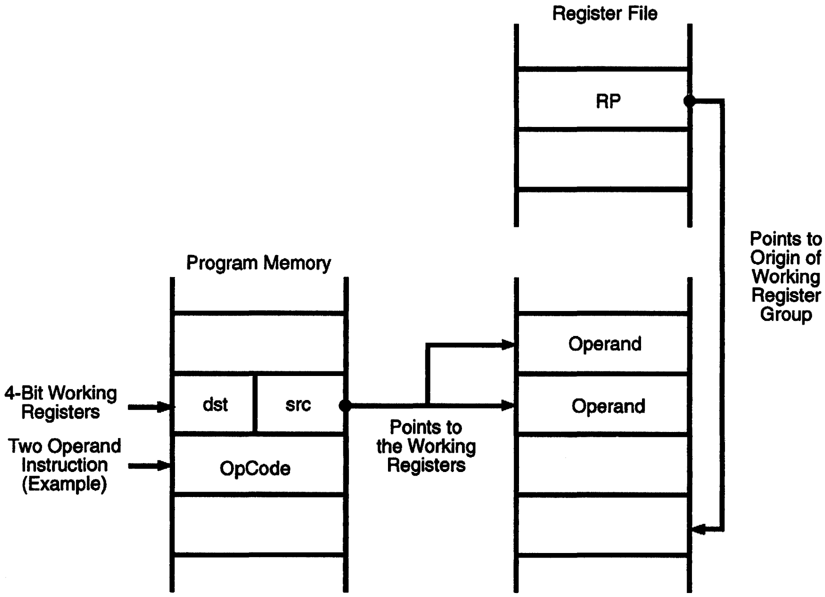

Register File
RP
Points to Origin of Working Register Group
Program Memory
4-Bit Working Registers
Two Operand Instruction (Example)
dst
src
OpCode
Points to the Working Registers
Operand
Operand

Figure 11-2. 4-Bit Register Addressing

11.3 Z8 INDIRECT REGISTER ADDRESSING (IR)

In the Indirect Register Addressing Mode, the contents of the specified register are equivalent to the address of the operand (Figures 11-3 and 11-4).

Depending upon the instruction selected, the specified register contents points to a Register, Program Memory, or an External Data Memory location.

When accessing program memory or External Data Memory, register pairs or Working Register pairs are used to hold the 16-bit addresses.

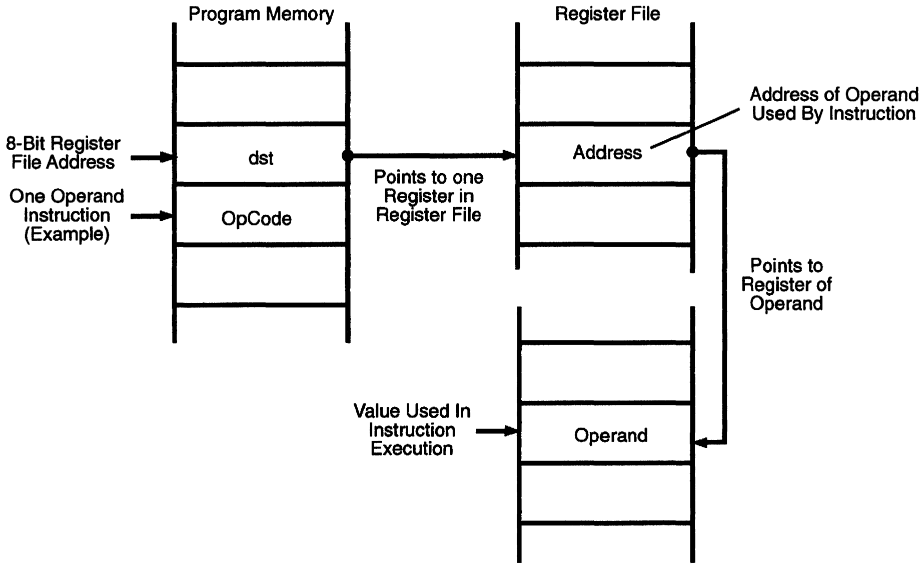

Program Memory
Register File
8-Bit Register File Address
One Operand Instruction (Example)
dst
OpCode
Points to one Register in Register File
Address
Address of Operand Used By Instruction
Points to Register of Operand
Value Used In Instruction Execution
Operand

Figure 11-3. Indirect Register Addressing to Register File

11.3 Z8 INDIRECT REGISTER ADDRESSING (IR) (Continued)

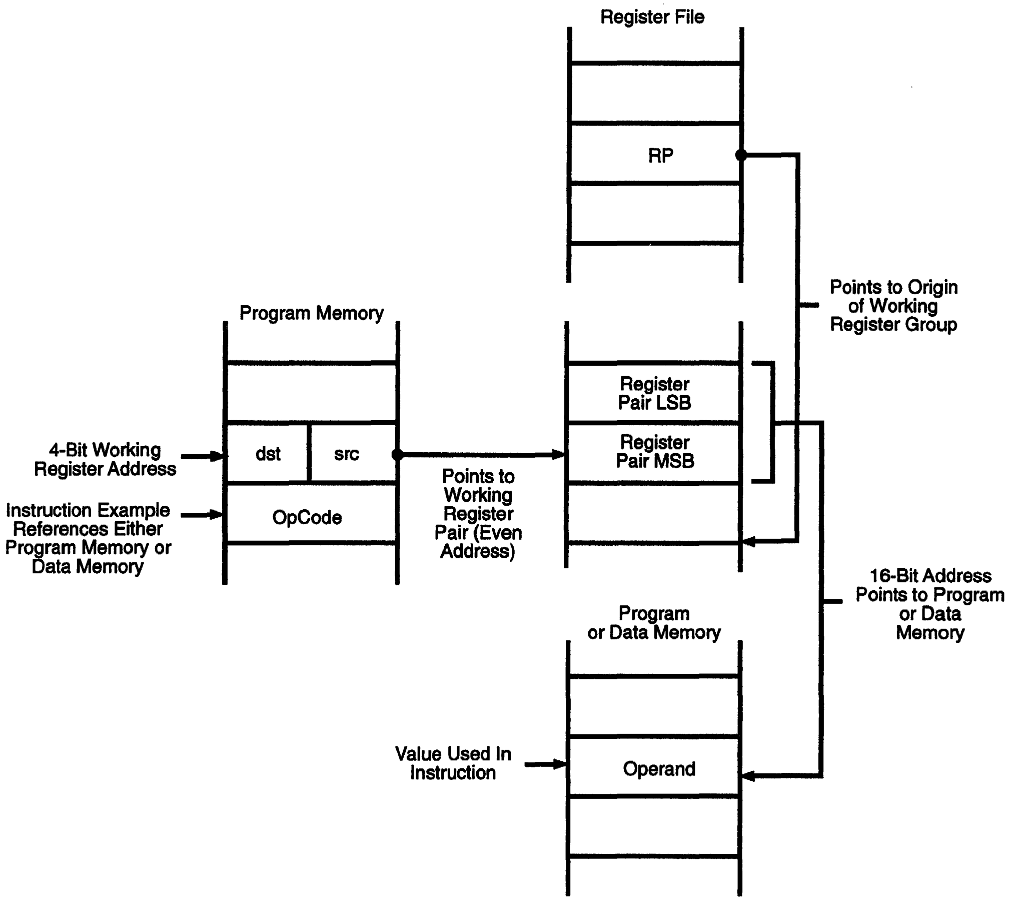

Register File
RP
Points to Origin of Working Register Group
Program Memory
4-Bit Working Register Address
dst
src
Instruction Example References Either Program Memory or Data Memory
OpCode
Points to Working Register Pair (Even Address)
Register Pair LSB
Register Pair MSB
16-Bit Address Points to Program or Data Memory
Program or Data Memory
Value Used In Instruction
Operand

Figure 11-4. Indirect Register Addressing to Program or Data Memory

11.4 Z8 INDEXED ADDRESSING (X)

The Indexed Addressing Mode is used only by the Load (LD) instruction. An indexed address consists of a register address offset by the contents of a designated Working Register (the Index). This offset is added to the register address to obtain the address of the operand. Figure 11-5 illustrates this addressing convention.

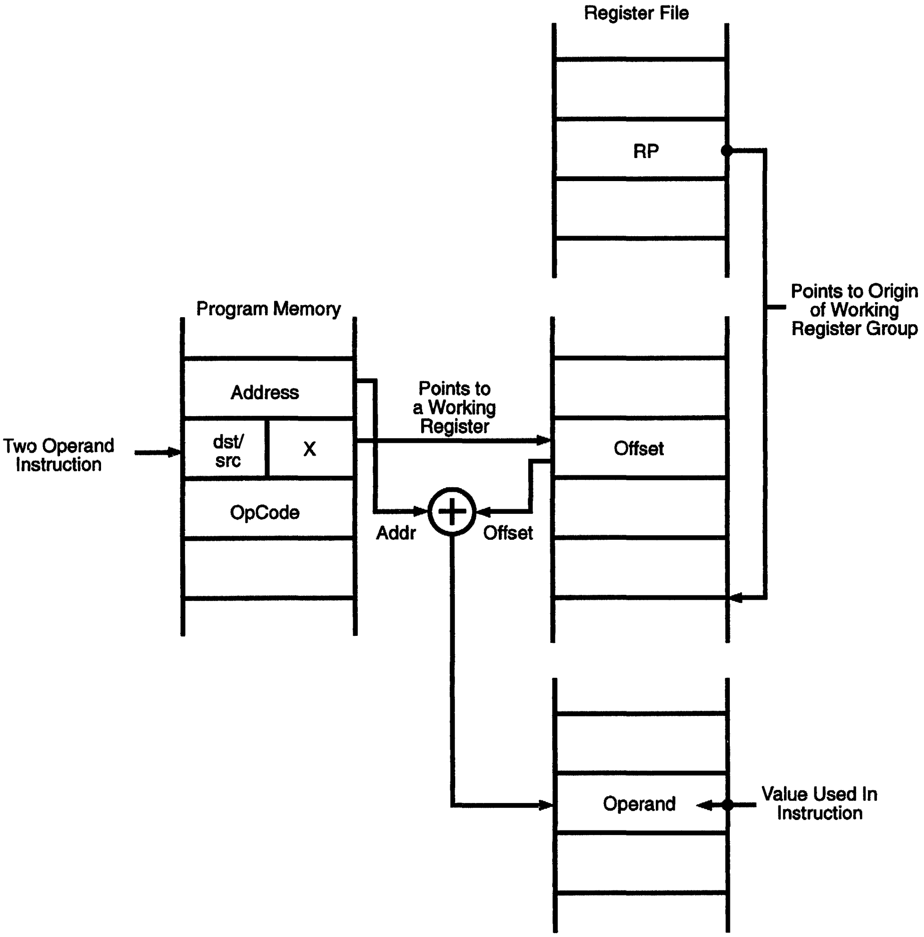


Figure 11-5. Indexed Register Addressing

11.5 Z8 DIRECT ADDRESSING (DA)

The Direct Addressing mode, as shown in Figure 11-6, specifies the address of the next instruction to be executed. Only the Conditional Jump (JP) and Call (CALL) instructions use this addressing mode.

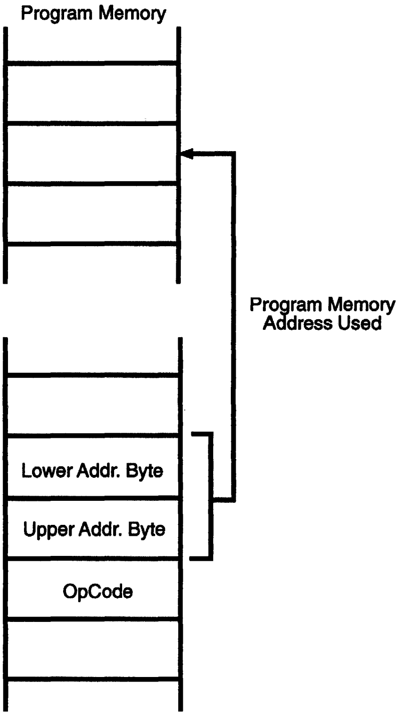


Figure 11-6. Direct Addressing

11.6 Z8 RELATIVE ADDRESSING (RA)

In the Relative Addressing mode, illustrated in Figure 11-7, the instruction specifies a two's-complement signed displacement in the range of −128 to +127. This is added to the contents of the PC to obtain the address of the next instruction to be executed. The PC (prior to the add) consists of the address of the instruction following the Jump Relative (JR) or Decrement and Jump if Non-Zero (DJNZ) instruction. JR and DJNZ are the only instructions which use this addressing mode.

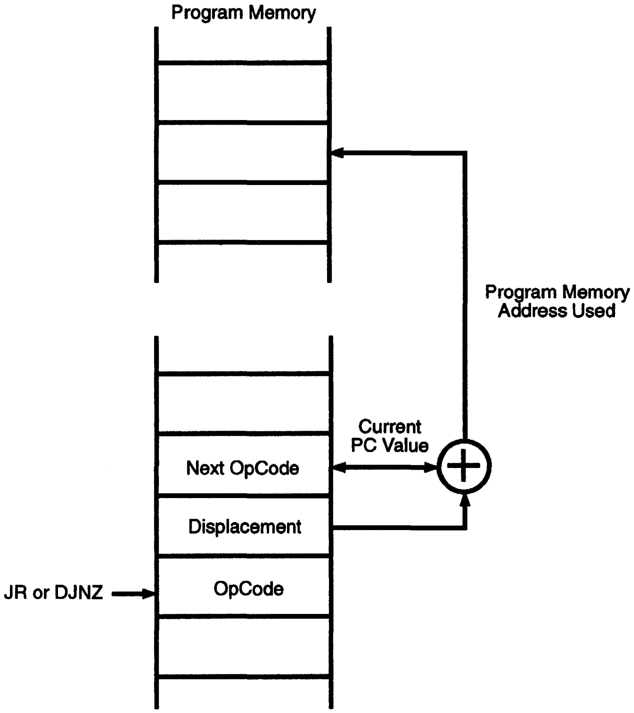


Figure 11-7. Relative Addressing

11.7 Z8 IMMEDIATE DATA ADDRESSING (IM)

Immediate data is considered an “addressing mode” for the purposes of this discussion. It is the only addressing mode that does not indicate a register or memory address as the source operand. The operand value used by the instruction is the value supplied in the operand field itself. Because an immediate operand is part of the instruction, it is always located in the Program Memory address space (Figure 11-8).

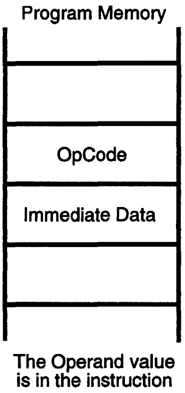


Figure 11-8. Immediate Data Addressing

CHAPTER 12
INSTRUCTION SET

12.1 Z8 FUNCTIONAL SUMMARY

Z8® instructions can be divided functionally into the following eight groups:

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

The following summary shows the instructions belonging to each group and the number of operands required for each. The source operand is 'src,' the destination operand is 'dst,'and a condition code is 'cc.'

Table 12-1. Load Instructions

Mnemonic	Operands	Instruction
CLR	dst	Clear
LD	dst,src	Load
LDC	dst,src	LoadConstant
LDE	dst,src	LoadExternal
POP	dst	Pop
PUSH	src	Push

Table 12-2. Arithmetic Instructions

Mnemonic	Operands	Instruction
ADC	dst, src	Add with Carry
ADD	dst, src	Add
CP	dst, src	Compare
DA	dst	Decimal Adjust
DEC	dst	Decrement
DECW	dst	Decrement Word
INC	dst	Increment
INCW	dst	Increment Word
SBC	dst, src	Subtract with Carry
SUB	dst, src	Subtract

Table 12-3. Logical Instructions

Mnemonic	Operands	Instruction
AND	dst,src	LogicalAND
COM	dst	Complement
OR	dst,src	LogicalOR
XOR	dst,src	LogicalExclusiveOR

Table 12-4. Program Control Instructions

Mnemonic	Operands	Instruction
CALL	dst	CallProcedure
DJNZ	dst,src	DecrementandJump Non-Zero
IRET		InterruptReturn
JP	cc,dst	Jump
JR	cc,dst	JumpRelative
RET		Return

Table 12-5. Bit Manipulation Instructions

Mnemonic	Operands	Instruction
TCM	dst,src	TestComplement UnderMask
TM	dst,src	TestUnderMask
AND	dst,src	BitClear
OR	dst,src	BitSet
XOR	dst,src	BitComplement

Table 12-6. Block Transfer Instructions

Mnemonic	Operands	Instruction
LDCI	dst,src	LoadConstant AutoIncrement
LDEI	dst,src	LoadExternal AutoIncrement

12.1 Z8 FUNCTIONAL SUMMARY (Continued)

Table 12-7. Rotate and Shift Instructions

Mnemonic	Operands	Instruction
RL	dst	Rotate Left
RLC	dst	Rotate Left Through Carry
RR	dst	Rotate Right
RRC	dst	Rotate Right Through Carry
SRA	dst	Shift Right Arithmetic
SWAP	dst	Swap Nibbles

Table 12-8. CPU Control Instructions

Mnemonic	Operands	Instruction
CCF		Complement Carry Flag
DI		Disable Interrupts
EI		Enable Interrupts
HALT		Halt
NOP		No Operation
RCF		Reset Carry Flag
SCF		Set Carry Flag
SRP	src	Set Register Pointer
STOP		Stop
WDH		WDT Enable During HALT
WDT		WDT Enable or Refresh

12.2 PROCESSOR FLAGS

The Flag Register (FCH) informs the user of the current status of the Z8. The flags and their bit positions in the Flag Register are shown in Figure 12-1.

The Z8 Flag Register contains six bits of status information which are set or cleared by CPU operations. Four of the bits (C, V, Z and S) can be tested for use with conditional Jump instructions. Two flags (H and D) cannot be tested and are used for BCD arithmetic. The two remaining bits in the Flag Register (F1 and F2) are available to the user, but they must be set or cleared by instructions and are not usable with conditional Jumps.

As with bits in the other control registers, the Flag Register bits can be set or reset by instructions; however, only those instructions that do not affect the flags as an outcome of the execution should be used (Load Immediate).

Note: The Watch-Dog Timer (WDT) instruction effects the Flags accordingly: Z=1, S=0, V=0.

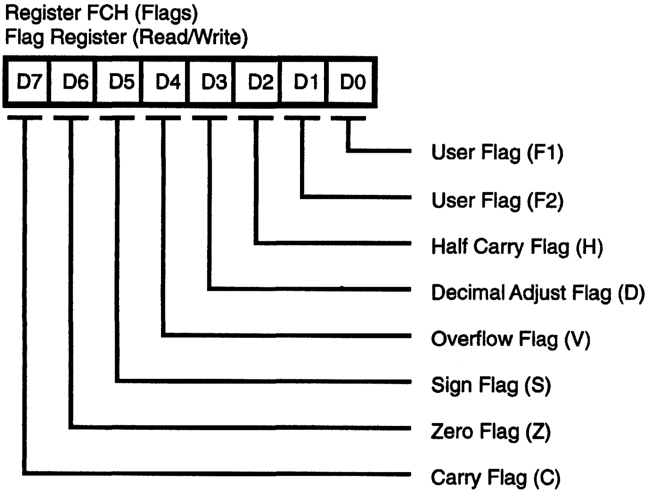


Figure 12-1. Z8 Flag Register

12.2.1 Carry Flag (C)

The Carry Flag is set to 1 whenever the result of an arithmetic operation generates a 'carry out of' or a 'borrow into' the high order bit 7. Otherwise, the Carry Flag is cleared to 0.

Following Rotate and Shift instructions, the Carry Flag contains the last value shifted out of the specified register.

An instruction can set, reset, or complement the Carry Flag.

IRET changes the value of the Carry Flag when the Flag Register saved in the Stack is restored.

12.2.2 Zero Flag (Z)

For arithmetic and logical operations, the Zero Flag is set to 1 if the result is zero. Otherwise, the Zero Flag is cleared to 0.

If the result of testing bits in a register is 00H, the Zero Flag is set to 1. Otherwise the Zero Flag is cleared to 0.

If the result of a Rotate or Shift operation is 00H, the Zero Flag is set to 1. Otherwise, the Zero Flag is cleared to 0.

IRET changes the value of the Zero Flag when the Flag Register saved in the Stack is restored. The WDT Instruction sets the Zero Flag to a 1.

12.2.3 Sign Flag (S)

The Sign Flag stores the value of the most significant bit of a result following an arithmetic, logical, Rotate, or Shift operation.

When performing arithmetic operations on signed numbers, binary two's-complement notation is used to represent and process information. A positive number is identified by a 0 in the most significant bit position (bit 7); therefore, the Sign Flag is also 0.

A negative number is identified by a 1 in the most significant bit position (bit 7); therefore, the Sign Flag is also 1.

IRET changes the value of the Sign Flag when the Flag Register saved in the Stack is restored.

12.2.4 Overflow Flag (V)

For signed arithmetic, Rotate, and Shift operations, the Overflow Flag is set to 1 when the result is greater than the maximum possible number (>127) or less than the minimum possible number (<–128) that can be represented in two's-complement form. The Overflow Flag is set to 0 if no overflow occurs.

Following logical operations the Overflow Flag is set to 0.

IRET changes the value of the Overflow Flag when the Flag Register saved in the Stack is restored.

12.2.5 Decimal Adjust Flag (D)

The Decimal Adjust Flag is used for BCD arithmetic. Since the algorithm for correcting BCD operations is different for addition and subtraction, this flag specifies what type of instruction was last executed so that the subsequent Decimal Adjust (DA) operation can function properly. Normally, the Decimal Adjust Flag cannot be used as a test condition.

After a subtraction, the Decimal Adjust Flag is set to 1. Following an addition it is cleared to 0.

IRET changes the value of the Decimal Adjust Flag when the Flag Register saved in the Stack is restored.

12.2.6 Half Carry Flag (H)

The Half Carry Flag is set to 1 whenever an addition generates a "carry out of" bit 3 (Overflow) or a subtraction generates a "borrow into" bit 3. The Half Carry Flag is used by the Decimal Adjust (DA) instruction to convert the binary result of a previous addition or subtraction into the correct decimal (BCD) result. As in the case of the Decimal Adjust Flag, the user does not normally access this flag.

IRET changes the value of the Half Carry Flag when the Flag Register saved in the Stack is restored.

12.3 CONDITION CODES

The C, Z, S, and V Flags control the operation of the 'Conditional' Jump instructions. Sixteen frequently useful functions of the flag settings are encoded in a 4-bit field called the condition code (cc), which forms bits 4-7 of the conditional instructions.

Table 12-9. Z8 Flag Definitions

Flag	Description
C	Carry Flag
Z	Zero Flag
S	Sign Flag
V	Overflow Flag
D	Decimal Adjust Flag
H	Half Carry Flag

Condition codes and flag settings are summarized in Tables 12-9, 12-10, and 12-11. Notation for the flags and how they are affected are as follows:

Table 12-10. Flag Settings Definitions

Symbol	Definition
0	Cleared to 0
1	Set to 1
*	Set or cleared according to operation
-	Unaffected
X	Undefined

Table 12-11. Condition Codes

Binary	Mnemonic	Definition	Flag Settings
0000	F	Always False	-
1000	(blank)	Always True	-
0111	C	Carry	C = 1
1111	NC	No Carry	C = 0
0110	Z	Zero	Z = 1
1110	NZ	Non-Zero	Z = 0
1101	PL	Plus	S = 0
0101	MI	Minus	S = 1
0100	OV	Overflow	V = 1
1100	NOV	No Overflow	V = 0
0110	EQ	Equal	Z = 1
1110	NE	Not Equal	Z = 0
1001	GE	Greater Than or Equal	(S XOR V) = 0
0001	LT	Less Than	(S XOR V) = 1
1010	GT	Greater Than	(Z OR (S XOR V)) = 0
0010	LE	Less Than or Equal	(Z OR (S XOR V)) = 1
1111	UGE	Unsigned Greater Than or Equal	C = 0
0111	ULT	Unsigned Less Than	C = 1
1011	UGT	Unsigned Greater Than	(C = 0 AND Z = 0) = 1
0011	ULE	Unsigned Less Than or Equal	(C OR Z) = 1

12.4 NOTATION AND BINARY ENCODING

In the detailed instruction descriptions that make up the rest of this chapter, operands and status flags are represented by a notational shorthand. Operands, condition codes, address modes, and their notations are as follows (Table 12-12):

Table 12-12. Notational Shorthand

Notation	Address Mode	Operand	Range *
cc	Condition Code		See condition codes
r	Working Register	Rn	n = 0 -15
R	Register or Working Register	Reg Rn	Reg. represents a number in the range of 00H to FFH n = 0 -15
RR	Register Pair or Working Register Pair	Reg RRp	Reg. represents an even number in the range of 00H to FEH p = 0, 2, 4, 6, 8, 10, 12, or 14
Ir	Indirect Working Register	@Rn	n = 0 -15
IR	Indirect Register or Indirect Working Register	@Reg @Rn	Reg. represents a number in the range of 00H to FFH n = 0- 15
Irr	Indirect Working Register Pair	@RRp	p = 0, 2, 4, 6, 8, 10, 12, or 14
IRR	Indirect Register Pair or Working Register Pair	@Reg @RRp	Reg. represents an even number in the range 00H to FFH p = 0, 2, 4, 6, 8, 10, 12, or 14
X	Indexed	Reg (Rn)	Reg. represents a number in the range of 00H to FFH and n = 0 - 15
DA	Direct Address	Addrs	Addrs. represents a number in the range of 00H to FFH
RA	Relative Address	Addrs	Addrs. represents a number in the range of +127 to −128 which is an offset relative to the address of the next instruction
IM	Immediate	#Data	Data is a number between 00H to FFH

* See the device product specification to determine the exact register file range available. The register file size varies by device type.

12.4 NOTATION AND BINARY ENCODING (Continued)

Additional symbols used are:

Table 12-13. Additional Symbols

Symbol	Definition
dst	Destination Operand
src	Source Operand
@	Indirect Address Prefix
SP	Stack Pointer
PC	Program Counter
FLAGS	Flag Register (FCH)
RP	Register Pointer (FDH)
IMR	Interrupt Mask Register (FBH)
#	Immediate Operand Prefix
%	Hexadecimal Number Prefix
H	Hexadecimal Number Suffix
B	Binary Number Suffix
OPC	Opcode

Assignment of a value is indicated by the symbol "←". For example,

dst ← dst + src

indicates the source data is added to the destination data and the result is stored in the destination location. The notation 'addr(n)' is used to refer to bit 'n' of a given location. For example,

dst (7)

refers to bit 7 of the destination operand.

12.4.1 Assembly Language Syntax

For proper instruction execution, Z8 assembly language syntax requires 'dst, src' be specified, in that order. The following instruction descriptions show the format of the object code produced by the assembler. This binary format should be followed by users who prefer manual program coding or who intend to implement their own assembler.

Example: If the contents of registers 43H and 08H are added and the result is stored in 43H, the assembly syntax and resulting object code is:

```
ASM:   ADD    43H,   08H    (ADD dst, src)
OBJ:   04     08     43     (OPC src, dst)
```

In general, whenever an instruction format requires an 8-bit register address, that address can specify any register location in the range 0 - 255 or a Working Register R0 - R15. If, in the above example, register 08H is a Working Register, the assembly syntax and resulting object code would be:

```
ASM:   ADD    43H,   R8     (ADD dst, src)
OBJ:   04     E8     43     (OPC src, dst)
```

Note: See the device product specification to determine the exact register file range available. The register file size varies by device type.

12.5 Z8 INSTRUCTION SUMMARY

Instruction and Operation	Address Mode dst	Address Mode src	Opcode Byte (Hex)	Flags Affected C Z S V D H
ADC dst, src	†		1[ ]	* * * * 0 *
dst←dst + src +C				
ADD dst, src	†		0[ ]	* * * * 0 *
dst←dst + src				
AND dst, src	†		5[ ]	- * * 0 - -
dst←dst AND src				
CALL dst	DA		D6	- - - - - - -
SP←SP - 2 and	IRR		D4	
PC←dst or				
@SP←PC				
CCF			EF	* - - - - - -
C←NOT C				
CLR dst	R		B0	- - - - - - -
dst←0	IR		B1	
COM dst	R		60	- * * 0 - -
dst←NOT dst	IR		61	
CP dst, src	†		A[ ]	* * * * - -
dst - src				
DA dst	R		40	* * * X - -
dst←DA dst	IR		41	
DEC dst	R		00	- * * * - -
dst←dst - 1	IR		01	
DECW dst	RR		80	- * * * - -
dst←dst - 1	IR		81	
DI			8F	- - - - - - -
IMR(7)←0				
DJNZ r, dst	RA		rA	- - - - - - -
r←r - 1			r = 0 - F	
if r ≠ 0, then				
PC←PC + dst				
Range: +127,				
−128				
EI			9F	- - - - - - -
IMR(7)←1				
HALT			7F	- - - - - - -
INC dst	r		rE	- * * * - -
dst←dst + 1			r = 0 - F	
	R		20	
	IR		21	

Instruction and Operation	Address Mode dst	Address Mode src	Opcode Byte (Hex)	Flags Affected C Z S V D H
INCW dst	RR		A0	- * * * - -
dst←dst + 1	IR		A1	
IRET			BF	* * * * * *
FLAGS←@SP;				
SP←SP + 1				
PC←@SP;				
SP←SP + 2, and				
IMR(7)←1				
JP cc, dst	DA		cD	- - - - - - -
if cc is true,			c = 0 - F	
then PC←dst	IRR		30	
JR cc, dst	RA		cB	- - - - - - -
if cc is true, then			c = 0 - F	
PC←PC + dst				
Range: +127 to −128				
LD dst, src	r	Im	rC	- - - - - - -
dst←src	r	R	r8	
	R	r	r9	
			r = 0 - F	
	r	X	C7	
	X	r	D7	
	r	Ir	E3	
	Ir	r	F3	
	R	R	E4	
	R	IR	E5	
	R	IM	E6	
	IR	IM	E7	
	IR	R	F5	
LDC dst, src	r	Irr	C2	- - - - - - -
dst←src	Irr	r	D2	
LDCI dst, src	Ir	Irr	C3	- - - - - - -
dst←src and	Irr	r	D3	
r←r + 1 or				
rr←rr + 1				
LDE dst, src	r	Irr	82	- - - - - - -
dst←src	Irr	r	92	
LDEI dst, src	r	Irr	C2	- - - - - - -
dst←src and	Irr	r	D2	
r ← r+1 or				
rr ←rr+1				
NOP			FF	- - - - - - -
OR dst, src	†		4[ ]	- * * 0 - -
dst←dst OR src				

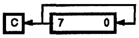
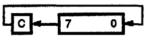
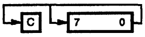
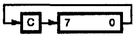
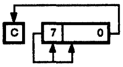
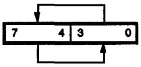
12.5 INSTRUCTION SUMMARY (Continued)

Instruction and Operation	Address Mode dst src	Opcode Byte (Hex)	Flags Affected C Z S V D H

POP dst
dst←@SP and
SP←SP + 1
	R
	IR
		50
		51
			- - - - - - -

PUSH src SP←SP - 1 and @SP←src	R IR	70 71	- - - - - - -
RCF C←0		CF	0 - - - - - -

RET
PC←@SP;
SP←SP + 2
		AF	- - - - - - -

RL dst 	R IR	90 91	* * * * - -
RLC dst 	R IR	10 11	* * * * - -
RR dst 	R IR	E0 E1	* * * * - -
RRC dst 	R IR	C0 C1	* * * * - -
SBC dst, src dst←dst - src - C	†	3[ ]	* * * * 1 *
SCF C←1		DF	1 - - - - -
SRA dst 	R IR	D0 D1	* * * 0 - -
SRP dst RP←src	Im	31	- - - - - - -
STOP		6F	- - - - - - -
SUB dst, src dst←dst - src	†	2[ ]	* * * * 1 *
SWAP dst 	R IR	F0 F1	X * * X - -
TCM dst, src (NOT dst) AND src	†	6[ ]	- * * 0 - -
TM dst, src dst AND src	†	7[ ]	- * * 0 - -
WDH		4F	- X X X - -
WDT		5F	- X X X - -
XOR dst, src dst←dst XOR src	†	B[ ]	- * * 0 - -

† These instructions have an identical set of addressing modes, which are encoded for brevity. The first Opcode nibble is found in the instruction set table above. The second nibble is expressed symbolically by a '[ ]' in this table, and its value is found in the following table to the left of the applicable addressing mode pair.

For example, the Opcode of an ADC instruction using the addressing modes r (destination) and Ir (source) is 13.

Address Mode dst	src	Lower Opcode Nibble
r	r	[2]
r	Ir	[3]
R	R	[4]
R	IR	[5]
R	IM	[6]
IR	IM	[7]

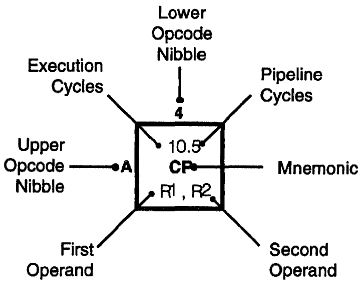
12.5.1 OPCODE MAP

Lower Nibble (Hex)

Upper Nibble (Hex)	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	6.5 **DEC** R1	6.5 **DEC** IR1	6.5 **ADD** r1, r2	6.5 **ADD** r1, Ir2	10.5 **ADD** R2, R1	10.5 **ADD** IR2, R1	10.5 **ADD** R1, IM	10.5 **ADD** IR1, IM	6.5 **LD** r1, R2	6.5 **LD** r2, R1	12/10.5 **DJNZ** r1, RA	12/10.0 **JR** cc, RA	6.5 **LD** r1, IM	12.10.0 **JP** cc, DA	6.5 **INC** r1	
1	6.5 **RLC** R1	6.5 **RLC** IR1	6.5 **ADC** r1, r2	6.5 **ADC** r1, Ir2	10.5 **ADC** R2, R1	10.5 **ADC** IR2, R1	10.5 **ADC** R1, IM	10.5 **ADC** IR1, IM	↓	↓	↓	↓	↓	↓	↓	
2	6.5 **INC** R1	6.5 **INC** IR1	6.5 **SUB** r1, r2	6.5 **SUB** r1, Ir2	10.5 **SUB** R2, R1	10.5 **SUB** IR2, R1	10.5 **SUB** R1, IM	10.5 **SUB** IR1, IM	↓	↓	↓	↓	↓	↓	↓	
3	8.0 **JP** IRR1	6.1 **SRP** IM	6.5 **SBC** r1, r2	6.5 **SBC** r1, Ir2	10.5 **SBC** R2, R1	10.5 **SBC** IR2, R1	10.5 **SBC** R1, IM	10.5 **SBC** IR1, IM	↓	↓	↓	↓	↓	↓	↓	
4	8.5 **DA** R1	8.5 **DA** IR1	6.5 **OR** r1, r2	6.5 **OR** r1, Ir2	10.5 **OR** R2, R1	10.5 **OR** IR2, R1	10.5 **OR** R1, IM	10.5 **OR** IR1, IM	↓	↓	↓	↓	↓	↓	↓	6.0 **WDH**
5	10.5 **POP** R1	10.5 **POP** IR1	6.5 **AND** r1, r2	6.5 **AND** r1, Ir2	10.5 **AND** R2, R1	10.5 **AND** IR2, R1	10.5 **AND** R1, IM	10.5 **AND** IR1, IM	↓	↓	↓	↓	↓	↓	↓	6.0 **WDT**
6	6.5 **COM** R1	6.5 **COM** IR1	6.5 **TCM** r1, r2	6.5 **TCM** r1, Ir2	10.5 **TCM** R2, R1	10.5 **TCM** IR2, R1	10.5 **TCM** R1, IM	10.5 **TCM** IR1, IM	↓	↓	↓	↓	↓	↓	↓	6.0 **STOP**
7	10/12.1 **PUSH** R2	12/14.1 **PUSH** IR2	6.5 **TM** r1, r2	6.5 **TM** r1, Ir2	10.5 **TM** R2, R1	10.5 **TM** IR2, R1	10.5 **TM** R1, IM	10.5 **TM** IR1, IM	↓	↓	↓	↓	↓	↓	↓	7.0 **HALT**
8	10.5 **DECW** RR1	10.5 **DECW** IR1	12.0 **LDE** r1, Irr2	18.0 **LDEI** Ir1, Irr2					↓	↓	↓	↓	↓	↓	↓	6.1 **DI**
9	6.5 **RL** R1	6.5 **RL** IR1	12.0 **LDE** r2, Irr1	18.0 **LDEI** Ir2, Irr1					↓	↓	↓	↓	↓	↓	↓	6.1 **EI**
A	10.5 **INCW** RR1	10.5 **INCW** IR1	6.5 **CP** r1, r2	6.5 **CP** r1, Ir2	10.5 **CP** R2, R1	10.5 **CP** IR2, R1	10.5 **CP** R1, IM	10.5 **CP** IR1, IM	↓	↓	↓	↓	↓	↓	↓	14.0 **RET**
B	6.5 **CLR** R1	6.5 **CLR** IR1	6.5 **XOR** r1, r2	6.5 **XOR** r1, Ir2	10.5 **XOR** R2, R1	10.5 **XOR** IR2, R1	10.5 **XOR** R1, IM	10.5 **XOR** IR1, IM	↓	↓	↓	↓	↓	↓	↓	16.0 **IRET**
C	6.5 **RRC** R1	6.5 **RRC** IR1	12.0 **LDC** r1, Irr2	18.0 **LDCI** Ir1, Irr2				10.5 **LD** r1,x,R2	↓	↓	↓	↓	↓	↓	↓	6.5 **RCF**
D	6.5 **SRA** R1	6.5 **SRA** IR1	12.0 **LDC** Irr1, r2	18.0 **LDCI** Irr1, Ir2	20.0 **CALL*** IRR1		20.0 **CALL** DA	10.5 **LD** r2,x,R1	↓	↓	↓	↓	↓	↓	↓	6.5 **SCF**
E	6.5 **RR** R1	6.5 **RR** IR1		6.5 **LD** r1, IR2	10.5 **LD** R2, R1	10.5 **LD** IR2, R1	10.5 **LD** R1, IM	10.5 **LD** IR1, IM	↓	↓	↓	↓	↓	↓	↓	6.5 **CCF**
F	8.5 **SWAP** R1	8.5 **SWAP** IR1		6.5 **LD** Ir1, r2		10.5 **LD** R2, IR1			↓	↓	↓	↓	↓	↓	↓	6.0 **NOP**

2 (columns 0–3) 3 (columns 4–7) 2 (columns 8–C) 3 (column D) 1 (columns E–F)

Bytes per Instruction



Legend:
R = 8-bit Address
r = 4-bit Address
R1 or r1 = Dst Address
R2 or r2 = Src Address

Sequence:
Opcode, First Operand,
Second Operand

Note: Blanks are reserved.

*2-byte instruction appears as a 3-byte instruction

12.6 INSTRUCTION DESCRIPTIONS AND FORMATS

ADC
ADD WITH CARRY

ADC
Add with Carry

ADC dst, src

Instruction Format:

		OPC	Address	Mode
	Cycles	(Hex)	dst	src
OPC | dst | src	6	12	r	r
	6	13	r	Ir
OPC | src | dst	10	14	R	R
	10	15	R	IR
OPC | dst | src	10	16	R	IM
	10	17	IR	IM

Operation: dst <— dst + src + C

The source operand, along with the setting of the Carry (C) Flag, is added to the destination operand. Two's complement addition is performed. The sum is stored in the destination operand. The contents of the source operand are not affected. In multiple precision arithmetic, this instruction permits the carry from the addition of low order operands to be carried into the addition of high order operands.

Flags:

C: Set if there is a carry from the most significant bit of the result; cleared otherwise.
Z: Set if the result is zero; cleared otherwise.
S: Set if the result is negative; cleared otherwise.
V: Set if an arithmetic overflow occurs, that is, if both operands are of the same sign and the result is of the opposite sign; cleared otherwise.
D: Always cleared.
H: Set if there is a carry from the most significant bit of the low order four bits of the result; cleared otherwise.

Note: Address modes R or IR can be used to specify a 4-bit Working Register. In this format, the source or destination Working Register operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.

E | src or E | dst

Example: If Working Register R3 contains 16H, the C Flag is set to 1, and Working Register R11 contains 20H, the statement:

ADC R3, R11
OpCode: 12 3B

leaves the value 37H in Working Register R3. The C, Z, S, V, D, and H Flags are all cleared.

ADC
ADD WITH CARRY

Example: If Working Register R16 contains 16H, the C Flag is not set, Working Register R10 contains 20H, and Register 20H contains 11H, the statement:

ADC R16, @R10
OpCode: 13 FA

leaves the value 27H in Working Register R16. The C, Z, S, V, D, and H Flags are all cleared.

Example: If Register 34H contains 2EH, the C Flag is set, and Register 12H contains 1BH, the statement:

ADC 34H, 12H
OpCode: 14 12 34

leaves the value 4AH in Register 34H. The H Flag is set, and the C, Z, S, V, and D Flags are cleared.

Example: If Register 4BH contains 82H, the C Flag is set, Working Register R3 contains 10H, and Register 10H contains 01H, the statement:

ADC 4BH, @R3
OpCode: 15 E3 4B

leaves the value 84H in Register 4BH. The S Flag is set, and the C, Z, V, D, and H Flags are cleared.

Example: If Register 6CH contains 2AH, and the C Flag is not set, the statement:

ADC 6CH, #03H
OpCode: 16 6C 03

leaves the value 2DH in Register 6CH. The C, Z, S, V, D, and H Flags are all cleared.

Example: If Register D4H contains 5FH, Register 5FH contains 4CH, and the C Flag is set, the statement:

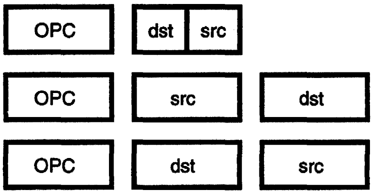
ADC @D4H, #02H
OpCode: 17 D4 02

leaves the value 4FH in Register 5FH. The C, Z, S, V, D, and H Flags are all cleared.

ADD
Add

ADD dst, src

Instruction Format:



Cycles	OPC (Hex)	Address dst	Mode src
6	02	r	r
6	03	r	Ir
10	04	R	R
10	05	R	IR
10	06	R	IM
10	07	IR	IM

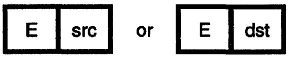
Operation: dst <— dst + src

The source operand is added to the destination operand. Two's complement addition is performed. The sum is stored in the destination operand. The contents of the source operand are not affected.

Flags:

C: Set if there is a carry from the most significant bit of the result; cleared otherwise.
Z: Set if the result is zero; cleared otherwise.
S: Set if the result is negative; cleared otherwise.
V: Set if an arithmetic overflow occurs, that is, if both operands are of the same sign and the result is of the opposite sign; cleared otherwise.
D: Always cleared.
H: Set if there is a carry from the most significant bit of the low order four bits of the result; cleared otherwise.

Note: Address modes R or IR can be used to specify a 4-bit Working Register. In this format, the source or destination Working Register operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.



Example: If Working Register R3 contains 16H and Working Register R11 contains 20H, the statement:

ADD R3, R11
OpCode: 02 3B

leaves the value 36H in Working Register R3. The C, Z, S, V, D, and H Flags are all cleared.

Example: If Working Register R16 contains 16H, Working Register R10 contains 20H, and Register 20H contains 11H, the statement:

ADD R16, @R10
OpCode: 03 FA

leaves the value 27H in Working Register R16. The C, Z, S, V, D, and H Flags are all cleared.

ADD
ADD

Example: If Register 34H contains 2EH and Register 12H contains 1BH, the statement:

ADD 34H, 12H
OpCode: 04 12 34

leaves the value 49H in Register 34H. The H Flag is set, and the C, Z, S, V, and D Flags are cleared.

Example: If Register 4BH contains 82H, Working Register R3 contains 10H, and Register 10H contains 01H, the statement:

ADD 3EH, @R3
OpCode: 05 E3 4B

leaves the value 83H in Register 4BH. The S Flag is set, and the C, Z, V, D, and H Flags are cleared.

Example: If Register 6CH contains 2AH, the statement:

ADD 6CH, #03H
OpCode: 06 6C 03

leaves the value 2DH in Register 6CH. The C, Z, S, V, D, and H Flags are all cleared.

Example: If Register D4H contains 5FH and Register 5FH contains 4CH, the statement:

ADD @D4H, #02H
OpCode: 07 D4 02

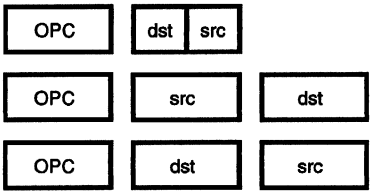
leaves the value 4EH in Register 5FH. The C, Z, S, V, D, and H Flags are all cleared.

AND
LOGICAL AND

AND
Logical AND

AND dst, src

Instruction Format:



Cycles	OPC (Hex)	Address dst	Mode src
6	52	r	r
6	53	r	Ir
10	54	R	R
10	55	R	IR
10	56	R	IM
10	57	IR	IM

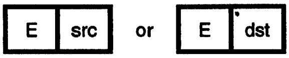
Operation: dst <— dst AND src

The source operand is logically ANDed with the destination operand. The AND operation results in a 1 being stored whenever the corresponding bits in the two operands are both 1, otherwise a 0 is stored. The result is stored in the destination operand. The contents of the source bit are not affected.

Flags:

C:	Unaffected
Z:	Set if the result is zero; cleared otherwise
S:	Set if the result of bit 7 is set; cleared otherwise
V:	Always reset to 0
D:	Unaffected
H:	Unaffected

Note: Address modes R or IR can be used to specify a 4-bit Working Register. In this format, the source or destination Working Register operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.



Example: If Working Register R1 contains 34H (00111000B) and Working Register R14 contains 4DH (10001101), the statement:

AND R1, R14
OpCode: 52 1E

leaves the value 04H (00001000) in Working Register R1. The Z, V, and S Flags are cleared.

Example: If Working Register R4 contains F9H (11111001B), Working Register R13 contains 7BH, and Register 7BH contains 6AH (01101010B), the statement:

AND R4, @R13
OpCode: 53 4D

leaves the value 68H (01101000B) in Working Register R4. The Z, V, and S Flags are cleared.

AND
LOGICAL AND

Example: If Register 3AH contains the value F5H (11110101B) and Register 42H contains the value 0AH (00001010), the statement:

AND 3AH, 42H
OpCode: 54 42 3A

leaves the value 00H (00000000B) in Register 3AH. The Z Flag is set, and the V and S Flags are cleared.

Example: If Working Register R5 contains F0H (11110000B), Register 45H contains 3AH, and Register 3AH contains 7FH (01111111B), the statement:

AND R5, @45H
OpCode: 55 45 E5

leaves the value 70H (01110000B) in Working Register R5. The Z, V, and S Flags are cleared.

Example: If Register 7AH contains the value F7H (11110111B), the statement:

AND 7AH, #F0H
OpCode: 56 7A F0

leaves the value F0H (11110000B) in Register 7AH. The S Flag is set, and the Z and V Flags are cleared.

Example: If Working Register R3 contains the value 3EH and Register 3EH contains the value ECH (11101100B), the statement:

AND @R3, #05H
OpCode: 57 E3 05

leaves the value 04H (00000100B) in Register 3EH. The Z, V, and S Flags are cleared.

CALL
CALL PROCEDURE

CALL
Call Procedure

CALL dst

Instruction Format:

		Cycles	OPC (Hex)	Address Mode dst
OPC	dst	20	D6	DA
OPC	dst	20	D4	IRR

Operation: SP <— SP - 2
@SP <— PC
PC <— dst

The Stack pointer is decremented by two, the current contents of the Program Counter (PC) (address of the first instruction following the CALL instruction) are pushed onto the top of the Stack, and the specified destination address is then loaded into the PC. The PC now points to the first instruction of the procedure.

At the end of the procedure a RET (return) instruction can be used to return to the original program flow. RET will pop the top of the Stack and replace the original value into the PC.

Flags:

C:	Unaffected
Z:	Unaffected
S:	Unaffected
V:	Unaffected
D:	Unaffected
H:	Unaffected

Note: Address mode IRR can be used to specify a 4-bit Working Register Pair. In this format, the destination Working Register Pair operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register Pair RR12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.

E	dst

Example: If the contents of the PC are 1A47H and the contents of the SP (Registers FEH and FFH) are 3002H, the statement:

CALL 3521H
OpCode: D6 35 21

causes the SP to be decremented to 3000H, 1A4AH (the address following the CALL instruction) to be stored in external data memory 3000 and 3001H, and the PC to be loaded with 3521H. The PC now points to the address of the first statement in the procedure to be executed.

CALL
CALL PROCEDURE

Example: If the contents of the PC are 1A47H, the contents of the SP (Register FFH) are 72H, the contents of Register A4H are 34H, and the contents of Register Pair 34H are 3521H, the statement:

CALL @A4H
OpCode: D4 A4

causes the SP to be decremented to 70H, 1A4AH (the address following the CALL instruction) to be stored in R70H and 71H, and the PC to be loaded with 3521H. The PC now points to the address of the first statement in the procedure to be executed.

CCF
COMPLEMENT CARRY FLAG

CCF
Complement Carry Flag

CCF

Instruction Format:

	Cycles	OPC (Hex)
OPC	6	EF

Operation: C <— NOT C

The C Flag is complemented. If C = 1, then it is changed to C = 0; or, if C = 0, then it is changed to C = 1.

Flags:

C:	Complemented
Z:	Unaffected
S:	Unaffected
V:	Unaffected
D:	Unaffected
H:	Unaffected

Example: If the C Flag contains a 0, the statement:

CCF
OpCode: EF

will change the C Flag from C = 0 to C = 1.

CLR
CLEAR

CLR
CLEAR

CLR dst

Instruction Format:

OPC | dst

Cycles	OPC (Hex)	Address Mode dst
6	B0	R
6	B1	IR

Operation: dst <— 0

The destination operand is cleared to 00H.

Flags:

C:	Unaffected
Z:	Unaffected
S:	Unaffected
V:	Unaffected
D:	Unaffected
H:	Unaffected

Note: Address modes R or IR can be used to specify a 4-bit Working Register. In this format, the destination Working Register operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.

E | dst

Example: If Working Register R6 contains AFH, the statement:

CLR R6
OpCode: B0 E6

will leave the value 00H in Working Register R6.

If Register A5H contains the value 23H, and Register 23H contains the value FCH, the statement:

CLR @A5H
OpCode: B1 A5

will leave the value 00H in Register 23H.

COM
COMPLEMENT

COM
Complement

COM dst

Instruction Format:

	Cycles	OPC (Hex)	Address Mode dst
OPC | dst	6	60	R
	6	61	IR

Operation: dst <— NOT dst

The contents of the destination operand are complemented (one's complement). All 1 bits are changed to 0, and all 0 bits are changed to 1.

Flags:

C: Unaffected
Z: Set if the result is zero; cleared otherwise.
S: Set if result bit 7 is set; cleared otherwise.
V: Always reset to 0.
D: Unaffected
H: Unaffected

Note: Address modes R or IR can be used to specify a 4-bit Working Register. In this format, the destination Working Register operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.

E	dst

Example: If Register 08H contains 24H (00100100B), the statement:

COM 08H
OpCode: 60 08

leaves the value DBH (11011011) in Register 08H. The S Flag is set, and the Z and V Flags are cleared.

Example: If Register 08H contains 24H, and Register 24H contains FFH (11111111B), the statement:

COM @08H
OpCode: 61 08

leaves the value 00H (00000000B) in Register 24H. The Z Flag is set, and the V and S Flags are cleared.

CP
COMPARE

CP
Compare

CP dst, src

Instruction Format:

		OPC	Address	Mode
	Cycles	(Hex)	dst	src
OPC dst src	6	A2	r	r
	6	A3	r	Ir
OPC src dst	10	A4	R	R
	10	A5	R	IR
OPC dst src	10	A6	R	IM
	10	A7	IR	IM

Operation: dst - src

The source operand is compared to (subtracted from) the destination operand, and the appropriate flags are set accordingly. The contents of both operands are unaffected.

Flags:

C: Cleared if there is a carry from the most significant bit of the result. Set otherwise indicating a borrow.
Z: Set if the result is zero; cleared otherwise.
S: Set if result bit 7 is set (negative); cleared otherwise.
V: Set if arithmetic overflow occurs; cleared otherwise.
D: Unaffected
H: Unaffected

Note: Address modes R or IR can be used to specify a 4-bit Working Register. In this format, the source or destination Working Register operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.

E src or E dst

Example: If Working Register R3 contains 16H and Working Register R11 contains 20H, the statement:

CP R3, R11
OpCode: A2 3B

sets the C and S Flags, and the Z and V Flags are cleared.

Example: If Working Register R15 contains 16H, Working Register R10 contains 20H, and Register 20H contains 11H, the statement:

CP R16, @R10
OpCode: A3 FA

clears the C, Z, S, and V Flags.

CP
COMPARE

Example: If Register 34H contains 2EH and Register 12H contains 1BH, the statement:

CP 34H,12H
OpCode: A4 12 34

clears the C, Z, S, and V Flags.

Example: If Register 4BH contains 82H, Working Register R3 contains 10H, and Register 10H contains 01H, the statement:

CP 4BH, @R3
OpCode: A5 E3 4B

sets the S Flag, and clears the C, Z, and V Flags.

Example: If Register 6CH contains 2AH, the statement:

CP 6CH, #2AH
OpCode: A6 6C 2A

sets the Z Flag, and the C, S, and V Flags are all cleared.

Example: If Register D4H contains FCH, and Register 5FH contains FCH, the statement:

CP @D4H, 7FH
OpCode: A7 D4 FF

sets the V Flag, and the C, Z, and S Flags are all cleared.

DA
DECIMAL ADJUST

DA
Decimal Adjust

DA dst

Instruction Format:

OPC | dst

Cycles	OPC (Hex)	Address Mode dst
8	40	R
8	41	IR

Operation: dst <— DA dst

The destination operand is adjusted to form two 4-bit BCD digits following a binary addition or subtraction operation on BCD encoded bytes. For addition (ADD and ADC) or subtraction (SUB and SBC), the following table indicates the operation performed.

Instruction	Carry Before DA	Bits 7-4 Value (HEX)	H Flag Before DA	Bits 3-0 Value (HEX)	Number Added To Byte	Carry After DA
	0	0-9	0	0-9	00	0
	0	0-8	0	A-F	06	0
	0	0-9	1	0-3	06	0
ADD	0	A-F	0	0-9	60	1
ADC	0	9-F	0	A-F	66	1
	0	A-F	1	0-3	66	1
	1	0-2	0	0-9	60	1
	1	0-2	0	A-F	66	1
	1	0-3	1	0-3	66	1
	0	0-9	0	0-9	00	0
SUB	0	0-8	1	6-F	FA	0
SBC	1	7-F	0	0-9	A0	1
	1	6-F	1	6-F	9A	1

If the destination operand is not the result of a valid addition or subtraction of BCD digits, the operation is undefined.

Flags:

C: Set if there is a carry from the most significant bit; cleared otherwise (see table above).
Z: Set if the result is zero; cleared otherwise.
S: Set if result bit 7 is set (negative); cleared otherwise.
V: Undefined
D: Unaffected
H: Unaffected

DA
DECIMAL ADJUST

Note: Address modes R or IR can be used to specify a 4-bit Working Register. In this format, the destination Working Register operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.

E	dst

Example: If addition is performed using the BCD value 15 and 27, the result should be 42. The sum is incorrect, however, when the binary representations are added in the destination location using standard binary arithmetic.

$$\begin{array}{rl} & 0001\ 0101 = 15H \\ + & \underline{0010}\ \underline{0111} = \underline{27H} \\ & 0011\ 1100 = 3CH \end{array}$$

If the result of the addition is stored in Register 5FH, the statement:

DA 5FH
OpCode: 40 5F

adjusts this result so the correct BCD representation is obtained.

$$\begin{array}{l} 0011\ 1100 = 3CH \\ \underline{0000}\ \underline{0110} = \underline{06H} \\ 0100\ 0010 = 42H \end{array}$$

Register 5F now contains the value 42H. The C, Z, and S Flags are cleared, and V is undefined.

Example: If addition is performed using the BCD value 15 and 27, the result should be 42. The sum is incorrect, however, when the binary representations are added in the destination location using standard binary arithmetic.

$$\begin{array}{rl} & 0001\ 0101 = 15H \\ + & \underline{0010}\ \underline{0111} = \underline{27H} \\ & 0011\ 1100 = 3CH \end{array}$$

If Register 45F contains the value 5FH, and the result of the addition is stored in Register 5FH, the statement:

DA @45H
OpCode: 40 45

adjusts this result so the correct BCD representation is obtained.

$$\begin{array}{l} 0011\ 1100 = 3CH \\ \underline{0000}\ \underline{0110} = \underline{06H} \\ 0100\ 0010 = 42H \end{array}$$

Register 5F now contains the value 42H. The C, Z, and S Flags are cleared, and V is undefined.

DEC
DECREMENT

DEC
Decrement

DEC dst

Instruction Format:

OPC	dst

Cycles	OPC (Hex)	Address Mode dst
6	00	R
6	01	IR

Operation: dst <— dst - 1

The contents of the destination operand are decremented by one.

Flags:

C: Unaffected
Z: Set if the result is zero; cleared otherwise
S: Set if the result of bit 7 is set (negative); cleared otherwise
V: Set if arithmetic overflow occurs; cleared otherwise
D: Unaffected
H: Unaffected

Note: Address modes R or IR can be used to specify a 4-bit Working Register. In this format, the destination Working Register operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.

E	dst

Example: If Working Register R10 contains 2A%, the statement:

DEC R10
OpCode: 00 EA

leaves the value 29H in Working Register R10. The Z, V, and S Flags are cleared.

Example: If Register B3H contains CBH, and Register CBH contains 01H, the statement:

DEC @B3H
OpCode: 01 B3

leaves the value 00H in Register CBH. The Z Flag is set, and the V and S Flags are cleared.

DECW
DECREMENT WORD

DECW
Decrement Word

DECW dst

Instruction Format:

OPC	dst

Cycles	OPC (Hex)	Address Mode dst
10	80	RR
10	81	IR

Operation: dst <— dst - 1

The contents of the destination (which must be an even address) operand are decremented by one. The destination operand can be a Register Pair or a Working Register Pair.

Flags:
C: Unaffected
Z: Set if the result is zero; cleared otherwise
S: Set if the result of bit 7 is set (negative); cleared otherwise
V: Set if arithmetic overflow occurs; cleared otherwise
D: Unaffected
H: Unaffected

Note: Address modes RR or IR can be used to specify a 4-bit Working Register Pair. In this format, the destination Working Register Pair operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register Pair R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.

E	dst

Example: If Register Pair 30H and 31H contain the value 0AF2H, the statement:

DECW 30H
OpCode: 80 30

leaves the value 0AF1H in Register Pair 30H and 31H. The Z, V, and S Flags are cleared.

Example: If Working Register R0 contains 30H and Register Pairs 30H and 31H contain the value FAF3H, the statement:

DECW @R0
OpCode: 81 E0

leaves the value FAF2H in Register Pair 30H and 31H. The S Flag is set, and the Z and V Flags are cleared.

DI
DISABLE INTERRUPTS

DI
Disable Interrupts

DI

Instruction Format:

	Cycles	OPC (Hex)
OPC	6	8F

Operation: IMR (7) <— 0

Bit 7 of Control Register FBH (the Interrupt Mask Register) is reset to 0. All interrupts are disabled, although they remain "potentially" enabled. (For instance, the Global Interrupt Enable is cleared, but not the individual interrupt level enables.)

Flags:

C:	Unaffected
Z:	Unaffected
S:	Unaffected
V:	Unaffected
D:	Unaffected
H:	Unaffected

Example: If Control Register FBH contains 8AH (10001010) (interrupts IRQ1 and IRQ3 are enabled), the statement:

DI
OpCode: 8F

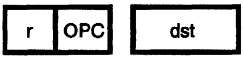
sets Control Register FBH to 0AH (00001010B) and disables these interrupts.

DJNZ
DECREMENT AND JUMP IF NONZERO

DJNZ
Decrement and Jump if Non-zero

DJNZ r, dst

Instruction Format:



Cycles		OPC (Hex)	Address Mode dst
12	if jump taken	rA	RA
10	if jump not taken	(r = 0 to F)	

Operation: r <— r - 1;
If r <> 0, PC <— PC + dst

The specified Working Register being used as a counter is decremented. If the contents of the specified Working Register are not zero after decrementing, then the relative address is added to the Program Counter (PC) and control passes to the statement whose address is now in the PC. The range of the relative address is +127 to –128. The original value of the PC is the address of the instruction byte following the DJNZ statement. When the specified Working Register counter reaches zero, control falls through to the statement following the DJNZ instruction.

Flags:

C:	Unaffected
Z:	Unaffected
S:	Unaffected
V:	Unaffected
D:	Unaffected
H:	Unaffected

Note: The Working Register being used as a counter must be one of the Registers from 04H to EFH. Use of one of the I/O ports, control or peripheral registers will have undefined results.

Example: DJNZ is typically used to control a "loop" of instructions. In this example, 12 bytes are moved from one buffer area in the register file to another. The steps involved are:

- Load 12 into the counter (Working Register R6).
- Set up the loop to perform the moves.
- End the loop with DJNZ.

The assembly listing required for this routine is as follows:

```
        LD R6, 12         ;Load Counter
LOOP:   LD R9, @R6        ;Move one byte to
        LD @R6, R9        ;new location
        DJNZ R6, LOOP     ;Decrement and Loop until
                          ;counter = 0
```

EI
ENABLE INTERRUPTS

EI
Enable Interrupts

EI

Instruction Format:

	Cycles	OPC (Hex)
OPC	6	9F

Operation: IMR (7) <— 0

Bit 7 of Control Register FBH (the Interrupt Mask Register) is set to 1. This allows potentially enabled interrupts to become enabled.

Flags:

C:	Unaffected
Z:	Unaffected
S:	Unaffected
V:	Unaffected
D:	Unaffected
H:	Unaffected

Example: If Control Register FBH contains 0AH (00001010) (interrupts IRQ1 and IRQ3 are selected), the statement:

EI
OpCode: 9F

sets Control Register FBH to 8AH (10001010B) and enables IRQ1 and IRQ3.

HALT
HALT

HALT
Halt

HALT

Instruction Format:

	Cycles	OPC (Hex)
OPC	6	7F

Operation: The HALT instruction turns off the internal CPU clock, but not the XTAL oscillation. The counter/timers and the external interrupts IRQ1, IRQ2, and IRQ3 remain active. The devices are recovered by interrupts, either externally or internally generated.

Flags:

C:	Unaffected
Z:	Unaffected
S:	Unaffected
V:	Unaffected
D:	Unaffected
H:	Unaffected

Note: In order to enter HALT mode, it is necessary to first flush the instruction pipeline to avoid suspending execution in mid-instruction. The user must execute a NOP immediately before the execution of the HALT instruction.

Example: Assuming the Z8 is in normal operation, the statements:

```
NOP
HALT
OpCodes: FF 7F
```

place the Z8 into HALT mode.

INC
INCREMENT

INC
Increment

INC dst

Instruction Format:

	Cycles	OPC (Hex)	Address Mode dst
$\boxed{\text{dst}}\boxed{\text{OPC}}$	6	rE	r
$\boxed{\text{OPC}}\ \boxed{\text{dst}}$	6	20	R
	6	21	IR

Operation: dst <— dst + 1

The contents of the destination operand are incremented by one.

Flags:

C: Unaffected
Z: Set if the result is zero; cleared otherwise.
S: Set if the result of bit 7 is set (negative); cleared otherwise.
V: Set if arithmetic overflow occurs; cleared otherwise.
D: Unaffected
H: Unaffected

Note: Address modes R or IR can be used to specify a 4-bit Working Register. In this format, the destination Working Register operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.

$\boxed{\text{E}}\boxed{\text{dst}}$

Example: If Working Register R10 contains 2AH, the statement:

INC R10
OpCode: AE

leaves the value 2BH in Working Register R10. The Z, V, and S Flags are cleared.

Example: If Register B3H contains CBH, the statement:

INC B3H
OpCode: 20 B3

leaves the value CCH in Register CBH. The S Flag is set, and the Z and V Flags are cleared.

Example: If Register B3H contains CBH and Register BCH contains FFH, the statement:

INC @B3H
OpCode: 21 B3

leaves the value 00H in Register CBH. The Z Flag is set, and the V and S Flags are cleared.

INCW
INCREMENT WORD

INCW
Increment Word

INCW dst

Instruction Format:

OPC	dst	Cycles	OPC (Hex)	Address Mode dst
		10	A0	RR
		10	A1	IR

Operation: dst <— dst - 1

The contents of the destination (which must be an even address) operand is decremented by one. The destination operand can be a Register Pair or a Working Register Pair.

Flags:

C: Unaffected
Z: Set if the result is zero; cleared otherwise.
S: Set if the result of bit 7 is set (negative); cleared otherwise.
V: Set if arithmetic overflow occurs; cleared otherwise.
D: Unaffected
H: Unaffected

Note: Address modes RR or IR can be used to specify a 4-bit Working Register Pair. In this format, the destination Working Register Pair operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register Pair R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.

E	dst

Example: If Register Pairs 30H and 31H contain the value 0AF2H, the statement:

INCW 30H
OpCode: A0 30

leaves the value 0AF3H in Register Pair 30H and 31H. The Z, V, and S Flags are cleared.

Example: If Working Register R0 contains 30H, and Register Pairs 30H and 31H contain the value FAF3H, the statement:

INCW @R0
OpCode: A1 E0

leaves the value FAF4H in Register Pair 30H and 31H. The S Flag is set, and the Z and V Flags are cleared.

IRET
INTERRUPT RETURN

IRET
Interrupt RETURN

IRET

Instruction Format:

	Cycles	OPC (Hex)
OPC	16	BF

Operation:

```
FLAGS <— @SP
    SP <— SP + 1
    PC <— @SP
    SP <— SP + 2
    IMR (7) <— 1
```

This instruction is issued at the end of an interrupt service routine. It restores the Flag Register (Control Register FCH) and the PC. It also re-enables any interrupts that are potentially enabled.

Flags:

C: Restored to original setting before the interrupt occurred.
Z: Restored to original setting before the interrupt occurred.
S: Restored to original setting before the interrupt occurred.
V: Restored to original setting before the interrupt occurred.
D: Restored to original setting before the interrupt occurred.
H: Restored to original setting before the interrupt occurred.

Example: If Stack Pointer Low Register FFH currently contains the value 45H, Register 45H contains the value 00H, Register 46H contains 6FH, and Register 47 Contains E4H, the statement:

IRET
OpCode: BF

restores the FLAG Register FCH with the value 00H, restores the PC with the value 6FE4H, re-enables the interrupts, and sets the Stack Pointer Low to 48H. The next instruction to be executed will be at location 6FE4H.

JP
JUMP

JP cc, dst

Instruction Format:

	Cycles		OPC (Hex)	Address Mode dst
cc OPC dst	12	if jump taken	ccD	DA
	10	if not taken	cc = 0 to F	
OPC dst	8		30	IRR

Operation: If cc (condition code) is true, then PC <— dst

A conditional jump transfers Program Control to the destination address if the condition specified by cc (condition code) is true. Otherwise, the instruction following the JP instruction is executed. See Section 12.3 for a list of condition codes.

The unconditional jump simply replaces the contents of the Program Counter with the contents of the register pair specified by the destination operand. Program Control then passes to the instruction addressed by the PC.

Flags:

C: Unaffected
Z: Unaffected
S: Unaffected
V: Unaffected
D: Unaffected
H: Unaffected

Note: Address mode IRR can be used to specify a 4-bit Working Register. In this format, the destination Working Register operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.

E	dst

Example: If the Carry Flag is set, the statement:

JP C, 1520H
OpCode: 7D 15 20

replaces the contents of the Program Counter with 1520H and transfers program control to that location. If the Carry Flag had not been set, control would have fallen through to the statement following the JP instruction.

Example: If Working Register Pair RR2 contains the value 3F45H, the statement:

JP @RR2
OpCode: 30 E2

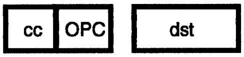
replaces the contents of the PC with the value 3F45H and transfers program control to that location.

JR
JUMP RELATIVE

JR
Jump Relative

JR cc, dst

Instruction Format:



Cycles		OPC (Hex)	Address Mode dst
10	if jump taken	ccB	RA
12	if jump not taken	cc = 0 to F	

Operation: If cc is true, PC <— PC + dst

If the condition specified by the "cc" is true, the relative address is added to the PC and control passes to the instruction located at the address specified by the PC (See Section 12.3 for a list of condition codes). Otherwise, the instruction following the JR instruction is executed. The range of the relative address is +127 to –128, and the original value of the PC is taken to be the address of the first instruction byte following the JR instruction.

Flags:

C:	Unaffected
Z:	Unaffected
S:	Unaffected
V:	Unaffected
D:	Unaffected
H:	Unaffected

Example: If the result of the last arithmetic operation executed is negative, the next four statements (which occupy a total of seven bytes) are skipped with the statement:

JR MI, #9
OpCode: 5B 09

If the result was not negative, execution would have continued with the instruction following the JR instruction.

Example: A short form of a jump –45 is:

JR #–45
OpCode: 8B D3

The condition code is "blank" in this case, and is assumed to be "always true."

LD
Load

LD dst, src

Instruction Format:

				OPC	Address Mode	
			Cycles	(Hex)	dst	src
dst OPC	src		6	rC	r	IM
			6	r8	r	R
src OPC	dst		6	r9 r = 0 to F	R*	r
OPC	dst src		6	E3	r	Ir
			6	F3	Ir	r
OPC	src	dst	10	E4	R	R
			10	E5	R	IR
OPC	dst	src	10	E6	R	IM
			10	E7	IR	IM
OPC	src	dst	10	F5	IR	R
OPC	dst X	src	10	C7	r	X
OPC	src X	dst	10	D7	X	r

* In this instance, only a full 8-bit register can be used.

Operation: dst <— src

The contents of the source operand are loaded into the destination operand. The contents of the source operand are not affected.

Flags:

C:	Unaffected
Z:	Unaffected
S:	Unaffected
V:	Unaffected
D:	Unaffected
H:	Unaffected

Note: Address modes R or IR can be used to specify a 4-bit Working Register. In this format, the source or destination Working Register operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.

E src or E dst

LD
LOAD

Example: The statement:

LD R15, #34H
OpCode: FC 34

loads the value 34H into Working Register R15.

Example: If Register 34H contains the value FCH, the statement:

LD R14, 34H
OpCode: F8 34

loads the value FCH into Working Register R15. The contents of Register 34H are not affected.

Example: If Working Register R14 contains the value 45H, the statement:

LD 34H, R14
OpCode: E9 34

loads the value 45H into Register 34H. The contents of Working Register R14 are not affected.

Example: If Working Register R12 contains the value 34H, and Register 34H contains the value FFH, the statement:

LD R13, @R12
OpCode: E3 DC

loads the value FFH into Working Register R13. The contents of Working Register R12 and Register R34 are not affected.

Example: If Working Register R13 contains the value 45H, and Working Register R12 contains the value 00H the statement:

LD @R13, R12
OpCode: F3 DC

loads the value 00H into Register 45H. The contents of Working Register R12 and Working Register R13 are not affected.

Example: If Register 45H contains the value CFH, the statement:

LD 34H, 45H
OpCode: E4 45 34

loads the value CFH into Register 34H. The contents of Register 45H are not affected.

Example: If Register 45H contains the value CFH and Register CFH contains the value FFH, the statement:

LD 34H, @45H
OpCode: E5 45 34

loads the value FFH into Register 34H. The contents of Register 45H and Register CFH are not affected.

Example: The statement:

LD 34H, #A4H
OpCode: E6 34 A4

loads the value A4H into Register 34H.

Example: If Working Register R14 contains the value 7FH, the statement:

LD @R14, #FCH
OpCode: E7 EE FC

loads the value FCH into Register 7FH. The contents of Working Register R14 are not affected.

Example: If Register 34H contains the value CFH and Register 45H contains the value FFH, the statement:

LD @34H, 45H
OpCode: F5 45 34

loads the value FFH into Register CFH. The contents of Register 34H and Register 45H are not affected.

Example: If Working Register R0 contains the value 08H and Register 2CH (24H + 08H = 2CH) contains the value 4FH, the statement:

LD R10, 24H(R0)
OpCode: C7 A0 24

loads Working Register R10 with the value 4FH. The contents of Working Register R0 and Register 2CH are not affected.

Example: If Working Register R0 contains the value 0BH and Working Register R10 contains 83H the statement:

LD F0H(R0), R10
OpCode: D7 A0 F0

loads the value 83H into Register FBH (F0H + 0BH = FBH). Since this is the Interrupt Mask Register, the LOAD statement has the effect of enabling IRQ0 and IRQ1. The contents of Working Registers R0 and R10 are unaffected by the load.

LDC
LOAD CONSTANT

LDC
Load Constant

LDC dst, src

Instruction Format:

			OPC	Address Mode	
		Cycles	(Hex)	src	dst
OPC	dst | src	12	C2	r	Irr
OPC	dst | src	12	D2	Irr	r

Operation: dst <— src

This instruction is used to load a byte constant from program memory into a Working Register, or vice versa. The address of the program memory location is specified by a Working Register Pair. The contents of the source operand are not affected.

Flags:

C:	Unaffected
Z:	Unaffected
S:	Unaffected
V:	Unaffected
D:	Unaffected
H:	Unaffected

Example: If Working Register Pair R6 and R7 contain the value 30A2H and program memory location 30A2H contains the value 22H, the statement:

LDC R2, @RR6
OpCode: C2 26

loads the value 22H into Working Register R2. The value of program memory location 30A2H is unchanged by the load.

Example: If Working Register R2 contains the value 22H, and Working Register Pair R6 and R7 contains the value 10A2H, the statement:

LDC @RR6, R2
OpCode: D2 26

loads the value 22H into program memory location 10A2H. The value of Working Register R2 is unchanged by the load.

Note: This instruction format is valid only for MCUs which can address external program memory.

LDCI
LOAD CONSTANT AUTO-INCREMENT

LDCI
Load Constant Auto-increment

LDCI dst, src

Instruction Format:

		Cycles	OPC (Hex)	Address Mode src	Address Mode dst
OPC	dst | src	18	C3	Ir	Irr
OPC	dst | src	18	D3	Irr	Ir

Operation: dst <— src
r <— r + 1
rr <— rr + 1

This instruction is used for block transfers of data between program memory and the Register File. The address of the program memory location is specified by a Working Register Pair, and the address of the Register File location is specified by Working Register. The contents of the source location are loaded into the destination location. Both addresses in the Working Registers are then incremented automatically. The contents of the source operand are not affected.

Flags:

C: Unaffected
Z: Unaffected
S: Unaffected
V: Unaffected
D: Unaffected
H: Unaffected

Example: If Working Register Pair R6-R7 contains 30A2H, program memory location 30A2H and 30A3H contain 22H and BCH respectively, and Working Register R2 contains 20H, the statement:

LDCI @R2, @RR6
OpCode: C3 26

loads the value 22H into Register 20H. Working Register Pair RR6 is incremented to 30A3H and Working Register R2 is incremented to 21H. A second

LDCI @R2, @RR6
OpCode: C3 26

loads the value BCH into Register 21H. Working Register Pair RR6 is incremented to 30A4H and Working Register R2 is incremented to 22H.

Note: This instruction format is valid only for MCUs which can address external program memory.

LDCI
LOAD CONSTANT AUTO-INCREMENT

Example: If Working Register R2 contains 20H, Register 20H contains 22H, Register 21H contains BCH, and Working Register Pair R6-R7 contains 30A2H, the statement:

LDCI @RR6, @R2
OpCode: D3 26

loads the value 22H into program memory location 30A2H. Working Register R2 is incremented to 21H and Working Register Pair R6-R7 is incremented to 30A3H. A second

LDCI @RR6, @R2
OpCode: D3 26

loads the value BCH into program memory location 30A3H. Working Register R2 is incremented to 22H and Working Register Pair R6-R7 is incremented to 30A4H.

LDE
Load External Data

LDE dst, src

Instruction Format:

			OPC	Address Mode	
		Cycles	(Hex)	src	dst
OPC	dst | src	12	82	r	Irr
OPC	src | dst	12	92	Irr	r

Operation: dst <— src

This instruction is used to load a byte from external data memory into a Working Register or vice versa. The address of the external data memory location is specified by a Working Register Pair. The contents of the source operand are not affected.

Flags:

C:	Unaffected
Z:	Unaffected
S:	Unaffected
V:	Unaffected
D:	Unaffected
H:	Unaffected

Example: If Working Register Pair R6 and R7 contain the value 40A2H and external data memory location 40A2H contains the value 22H, the statement:

LDE R2, @RR6
OpCode: 82 26

loads the value 22H into Working Register R2. The value of external data memory location 40A2H is unchanged by the load.

Example: If Working Register Pair R6 and R7 contain the value 404AH and Working Register R2 contains the value 22H, the statement:

LDE @RR6, R2
OpCode: 92 26

loads the value 22H into external data memory location 404AH

Note: This instruction format is valid only for MCUs which can address external data memory.

LDEI
LOAD EXTERNAL DATA AUTO-INCREMENT

LDEI
Load External Data Auto-increment

LDEI dst, src

Instruction Format:

			OPC	Address Mode	
		Cycles	(Hex)	src	dst
OPC	dst | src	18	83	Ir	Irr
OPC	src | dst	18	93	Irr	Ir

Operation: dst <— src
r <— r + 1
rr <— rr + 1

This instruction is used for block transfers of data between external data memory and the Register File. The address of the external data memory location is specified by a Working Register Pair, and the address of the Register File location is specified by a Working Register. The contents of the source location are loaded into the destination location. Both addresses in the Working Registers are then incremented automatically. The contents of the source are not affected.

Flags:

C:	Unaffected
Z:	Unaffected
S:	Unaffected
V:	Unaffected
D:	Unaffected
H:	Unaffected

Example: If Working Register Pair R6 and R7 contains 404AH, external data memory location 404AH and 404BH contain ABH and C3H respectively, and Working Register R2 contains 22H, the statement:

LDEI @R2, @RR6
OpCode: 83 26

loads the value ABH into Register 22H. Working Register Pair RR6 is incremented to 404BH and Working Register R2 is incremented to 23H. A second

LDCI @R2, @RR6
OpCode: 83 26

loads the value C3H into Register 23H. Working Register Pair RR6 is incremented to 404CH and Working Register R2 is incremented to 24H.

LDEI
LOAD EXTERNAL DATA AUTO-INCREMENT

Example: If Working Register R2 contains 22H, Register 22H contains ABH, Register 23H contains C3H, and Working Register Pair R6 and R7 contains 404AH, the statement:

LDEI @RR6, @R2
OpCode: 93 26

loads the value ABH into external data memory location 404AH. Working Register R2 is incremented to 23H and Working Register Pair RR6 is incremented to 404BH. A second

LDCI @RR6, @R2
OpCode: 93 26

loads the value C3H into external data memory location 404BH. Working Register R2 is incremented to 24H and Working Register Pair RR6 is incremented to 404CH.

Note: This instruction format is valid only for MCUs which can address external data memory.

NOP
NO OPERATION

NOP
No Operation

NOP

Instruction Format:

	Cycles	OPC (Hex)
OPC	6	FF

Operation: No action is performed by this instruction. It is typically used for timing delays or clearing the pipeline.

Flags:

C:	Unaffected
Z:	Unaffected
S:	Unaffected
V:	Unaffected
D:	Unaffected
H:	Unaffected

OR
LOGICAL OR

OR
Logical OR

OR dst, src

Instruction Format:

	Cycles	OPC	Address Mode	
		(Hex)	dst	src
OPC | dst | src	6	42	r	r
	6	43	r	Ir
OPC | src | dst	10	44	R	R
	10	45	R	IR
OPC | dst | src	10	46	R	IM
	10	47	IR	IM

Operation: dst <— dst OR src

The source operand is logically ORed with the destination operand and the result is stored in the destination operand. The contents of the source operand are not affected. The OR operation results in a one bit being stored whenever either of the corresponding bits in the two operands is a one. Otherwise, a zero bit is stored.

Flags:

C:	Unaffected
Z:	Set if the result is zero; cleared otherwise
S:	Set if the result of bit 7 is set; cleared otherwise
V:	Always reset to 0
D:	Unaffected
H:	Unaffected

Note: Address modes R or IR can be used to specify a 4-bit Working Register. In this format, the source or destination Working Register operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.

E | src or E | dst

Example: If Working Register R1 contains 34H (00111000B) and Working Register R14 contains 4DH (10001101), the statement:

OR R1, R14
OpCode: 42 1E

leaves the value BDH (10111101B) in Working Register R1. The S Flag is set, and the Z and V Flags are cleared.

Example: If Working Register R4 contains F9H (11111001B), Working Register R13 contains 7BH, and Register 7B contains 6AH (01101010B), the statement:

OR R4, @R13
OpCode: 43 4D

leaves the value FBH (11111011B) in Working Register R4. The S Flag is set, and the Z and V Flags are cleared.

OR
LOGICAL OR

Example: If Register 3AH contains the value F5H (11110101B) and Register 42H contains the value 0AH (00001010), the statement:

OR 3AH, 42H
OpCode: 44 42 3A

leaves the value FFH (11111111B) in Register 3AH. The S Flag is set, and the Z and V Flags are cleared.

Example: If Working Register R5 contains 70H (01110000B), Register 45H contains 3AH, and Register 3AH contains 7FH (01111111B), the statement:

OR R5, @45H
OpCode: 45 45 E5

leaves the value 7FH (01111111B) in Working Register R5. The Z, V, and S Flags are cleared.

Example: If Register 7AH contains the value F3H (11110111B), the statement:

OR 7AH, #F0H
OpCode: 46 7A F0

leaves the value F3H (11110111B) in Register 7AH. The S Flag is set, and the Z and V Flags are cleared.

Example: If Working Register R3 contains the value 3EH and Register 3EH contains the value 0CH (00001100B), the statement:

OR @R3, #05H
OpCode: 57 E3 05

leaves the value 0DH (00001101B) in Register 3EH. The Z, V, and S Flags are cleared.

POP
Pop

POP dst

Instruction Format:

		Cycles	OPC (Hex)	Address Mode dst
OPC	dst	10	50	R
		10	51	IR

Operation: dst <— @SP
SP <— SP + 1

The contents of the location specified by the SP (Stack Pointer) are loaded into the destination operand. The SP is then incremented automatically.

Flags:

C:	Unaffected
Z:	Unaffected
S:	Unaffected
V:	Unaffected
D:	Unaffected
H:	Unaffected

Note: Address modes R or IR can be used to specify a 4-bit Working Register. In this format, the destination Working Register operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.

E	dst

Example: If the SP (Control Registers FEH and FFH) contains the value 70H and Register 70H contains 44H, the statement:

POP 34H
OpCode: 50 34

loads the value 44H into Register 34H. After the POP operation, the SP contains 71H. The contents of Register 70 are not affected.

Example: If the SP (Control Registers FEH and FFH) contains the value 1000H, external data memory location 1000H contains 55H, and Working Register R6 contains 22H, the statement:

POP @R6
OpCode: 51 E6

loads the value 55H into Register 22H. After the POP operation, the SP contains 1001H. The contents of Working Register R6 are not affected.

PUSH
PUSH

PUSH
Push

PUSH src

Instruction Format:

OPC	src

Cycles		OPC (Hex)	Address Mode dst
10	Internal Stack	70	R
12	External Stack		
12	Internal Stack	71	IR
14	External Stack		

Operation: SP <— SP - 1
@SP <— src

The contents of the SP (stack pointer) are decremented by one, then the contents of the source operand are loaded into the location addressed by the decremented SP, thus adding a new element to the stack.

Flags:
C: Unaffected
Z: Unaffected
S: Unaffected
V: Unaffected
D: Unaffected
H:
Unaffected

Note: Address modes R or IR can be used to specify a 4-bit Working Register. In this format, the destination Working Register operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.

E	dst

Example: If the SP contains 1001H, the statement:

PUSH FCH
OpCode: 70 FC

stores the contents of Register FCH (the Flag Register) in location 1000H. After the PUSH operation, the SP contains 1000H.

Example: If the SP contains 61H and Working Register R4 contains FCH, the statement:

PUSH @R4
OpCode: 71 E4

stores the contents of Register FCH (the Flag Register) in location 60H. After the PUSH operation, the SP contains 60H.

RCF
RESET CARRY FLAG

RCF
Reset Carry Flag

RCF

Instruction Format:

	Cycles	OPC (Hex)
OPC	6	CF

Operation: C <— 0

The C Flag is reset to 0, regardless of its previous value.

Flags:

C:	Reset to 0
Z:	Unaffected
S:	Unaffected
V:	Unaffected
D:	Unaffected
H:	Unaffected

Example: If the C Flag is currently set, the statement:

RCF
OpCode: CF

resets the Carry Flag to 0.

RET
RETURN

RET
Return

RET

Instruction Format:

	Cycles	OPC (Hex)
OPC	14	AF

Operation: PC <— @SP
SP <— SP + 2

This instruction is normally used to return from a procedure entered by a CALL instruction. The contents of the location addressed by the SP are popped into the PC. The next statement executed is the one addressed by the new contents of the PC. The stack pointer is also incremented by two.

Flags:

C:	Unaffected
Z:	Unaffected
S:	Unaffected
V:	Unaffected
D:	Unaffected
H:	Unaffected

Note: Each PUSH instruction executed within the subroutine should be countered with a POP instruction in order to guarantee the SP is at the correct location when the RET instruction is executed. Otherwise the wrong address will be loaded into the PC and the program will not operate as desired.

Example: If SP contains 2000H, external data memory location 2000H contains 18H, and location 2001H contains B5H, the statement:

RET
OpCode: AF

leaves the value 2002H in the SP, and the PC contains 18B5H, the address of the next instruction to be executed.

RL
ROTATE LEFT

RL
Rotate Left

RL dst

Instruction Format:

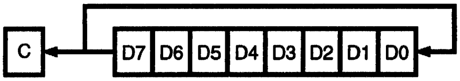
OPC | dst

Cycles	OPC (Hex)	Address Mode dst
6	90	R
6	91	IR

Operation:

```
C <— dst(7)
dst(0) <— dst(7)
dst(1) <— dst(0)
dst(2) <— dst(1)
dst(3) <— dst(2)
dst(4) <— dst(3)
dst(5) <— dst(4)
dst(6) <— dst(5)
dst(7) <— dst(6)
```

The contents of the destination operand are rotated left by one bit position. The initial value of bit 7 is moved to the bit 0 position and also into the Carry Flag.



Flags:

C: Set if the bit rotated from the most significant bit position was 1 (i.e., bit 7 was 1).
Z: Set if the result is zero; cleared otherwise.
S: Set if the result in bit 7 is set; cleared otherwise.
V: Set if arithmetic overflow occurred (if the sign of the destination operand changed during rotation); cleared otherwise.
D: Unaffected
H: Unaffected

Note:

Address modes R or IR can be used to specify a 4-bit Working Register. In this format, the destination Working Register operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.

E | dst

RL
ROTATE LEFT

Example: If the contents of Register C6H are 88H (10001000B), the statement:

RL C6H
OpCode: 80 C6

leaves the value 11H (00010001B) in Register C6H. The C and V Flags are set, and the S and Z Flags are cleared.

Example: If the contents of Register C6H are 88H, and the contents of Register 88H are 44H (01000100B), the statement:

RL @C6H
OpCode: 81 C6

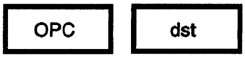
leaves the value 88H in Register 88H (10001000B). The S and V Flags are set, and the C and Z Flags are cleared.

RLC
ROTATE LEFT THROUGH CARRY

RLC
Rotate Left Through Carry

RLC dst

Instruction Format:

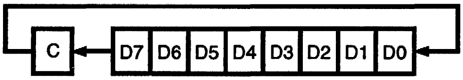


Cycles	OPC (Hex)	Address Mode dst
6	10	R
6	11	IR

Operation:

C <— dst(7)
dst(0) <— C
dst(1) <— dst(0)
dst(2) <— dst(1)
dst(3) <— dst(2)
dst(4) <— dst(3)
dst(5) <— dst(4)
dst(6) <— dst(5)
dst(7) <— dst(6)

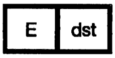
The contents of the destination operand along with the C Flag are rotated left by one bit position. The initial value of bit 7 replaces the C Flag and the initial value of the C Flag replaces bit 0.



Flags:

C: Set if the bit rotated from the most significant bit position was 1 (i.e., bit 7 was 1).
Z: Set if the result is zero; cleared otherwise.
S: Set if the result bit 7 is set; cleared otherwise.
V: Set if arithmetic overflow occurred (if the sign of the destination operand changed during rotation); cleared otherwise.
D: Unaffected
H: Unaffected

Note: Address modes R or IR can be used to specify a 4-bit Working Register. In this format, the destination Working Register operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.



RLC
ROTATE LEFT THROUGH CARRY

Example: If the C Flag is reset and Register C6 contains 8F (10001111B), the statement:

RLC C6
OpCode: 10 C6

leaves Register C6 with the value 1EH (00011110B). The C and V Flags are set, and S and Z Flags are cleared.

Example: If the C Flag is reset, Working Register R4 contains C6H, and Register C6 contains 8F (10001111B), the statement:

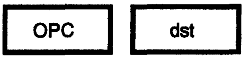
RLC @R4
OpCode: 11 E4

leaves Register C6 with the value 1EH (00011110B). The C and V Flags are set, and S and Z Flags are cleared.

RR
Rotate Right

RR dst

Instruction Format:

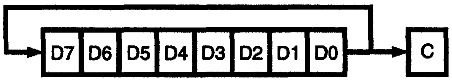


Cycles	OPC (Hex)	Address Mode dst
6	E0	R
6	E1	IR

Operation:

C <— dst(0)
dst(0) <— dst(1)
dst(1) <— dst(2)
dst(2) <— dst(3)
dst(3) <— dst(4)
dst(4) <— dst(5)
dst(5) <— dst(6)
dst(6) <— dst(7)
dst(7) <— dst(0)

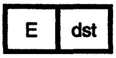
The contents of the destination operand are rotated to the right by one bit position. The initial value of bit 0 is moved to bit 7 and also into the C Flag.



Flags:

C: Set if the bit rotated from the least significant bit position was 1 (i.e., bit 0 was 1).
Z: Set if the result is zero; cleared otherwise.
S: Set if the result bit 7 is set; cleared otherwise.
V: Set if arithmetic overflow occurred (if the sign of the destination operand changed during rotation); cleared otherwise.
D: Unaffected
H: Unaffected

Note: Address modes R or IR can be used to specify a 4-bit Working Register. In this format, the destination Working Register operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.



RR
ROTATE RIGHT

Example: If the contents of Working Register R6 are 31H (00110001B), the statement:

RR R6
OpCode: E0 E6

leaves the value 98H (10011000) in Working Register R6. The C, V, and S Flags are set, and the Z Flag is cleared.

Example: If the contents of Register C6 are 31H and the contents of Register 31H are 7EH (01111110B), the statement:

RR @C6
OpCode: E1 C6

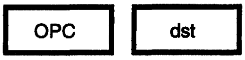
leaves the value 4FH (00111111) in Register 31H. The C, Z, V, and S Flags are cleared.

RRC
ROTATE RIGHT THROUGH CARRY

RRC
Rotate Right Through Carry

RRC dst

Instruction Format:

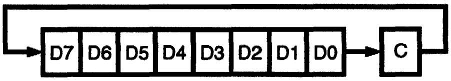


Cycles	OPC (Hex)	Address Mode dst
6	C0	R
6	C1	IR

Operation:

C <— dst(0)
dst(0) <— dst(1)
dst(1) <— dst(2)
dst(2) <— dst(3)
dst(3) <— dst(4)
dst(4) <— dst(5)
dst(5) <— dst(6)
dst(6) <— dst(7)
dst(7) <— C

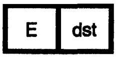
The contents of the destination operand with the C Flag are rotated right by one bit position. The initial value of bit 0 replaces the C Flag and the initial value of the C Flag replaces bit 7.



Flags:

C: Set if the bit rotated from the least significant bit position was 1 (i.e., bit 0 was 1).
Z: Set if the result is zero; cleared otherwise.
S: Set if the result bit 7 is set; cleared otherwise.
V: Set if arithmetic overflow occurred (if the sign of the destination operand changed during rotation); cleared otherwise.
D: Unaffected
H: Unaffected

Note: Address modes R or IR can be used to specify a 4-bit Working Register. In this format, the destination Working Register operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.



RRC
ROTATE RIGHT THROUGH CARRY

Example: If the contents of Register C6H are DDH (11011101B) and the C Flag is reset, the statement:

RRC C6H
OpCode: C0 C6

leaves the value 6EH (01101110B) in register C6H. The C and V Flags are set, and the Z and S Flags are cleared.

Example: If the contents of Register 2C are EDH, the contents of Register EDH is 00H (00000000B), and the C Flag is reset, the statement:

RRC @2CH
OpCode: C1 2C

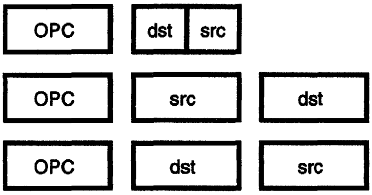
leaves the value 01H (00000001B) in Register EDH. The C, Z, S, and V Flags are reset.

SBC
SUBTRACT WITH CARRY

SBC
Subtract With Carry

SBC dst, src

Instruction Format:



Cycles	OPC (Hex)	Address dst	Mode src
6	32	r	r
6	33	r	Ir
10	34	R	R
10	35	R	IR
10	36	R	IM
10	37	IR	IM

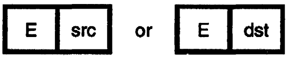
Operation: dst <— dst - src - C

The source operand, along with the setting of the C Flag, is subtracted from the destination operand and the result is stored in the destination operand. The contents of the source operand are not affected. Subtraction is performed by adding the two's complement of the source operand to the destination operand. In multiple precision arithmetic, this instruction permits the carry (borrow) from the subtraction of low order operands to be subtracted from the subtraction of high order operands.

Flags:

C: Cleared if there is a carry from the most significant bit of the result; set otherwise, indicating a "borrow."
Z: Set if the result is 0; cleared otherwise.
V: Set if arithmetic overflow occurred (if the operands were of opposite sign and the sign of the result is the same as the sign of the source); reset otherwise.
S: Set if the result is negative; cleared otherwise.
H: Cleared if there is a carry from the most significant bit of the low order four bits of the result; set otherwise indicating a "borrow."
D: Always set to 1.

Note: Address modes R or IR can be used to specify a 4-bit Working Register. In this format, the source or destination Working Register operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.



Example: If Working Register R3 contains 16H, the C Flag is set to 1, and Working Register R11 contains 20H, the statement:

SBC R3, R11
OpCode: 32 3B

leaves the value F5H in Working Register R3. The C, S, and D Flags are set, and the Z, V, and H Flags are all cleared.

SBC
SUBTRACT WITH CARRY

Example: If Working Register R15 contains 16H, the C Flag is not set, Working Register R10 contains 20H, and Register 20H contains 11H, the statement:

SBC R16, @R10
OpCode: 33 FA

leaves the value 05H in Working Register R15. The D Flag is set, and the C, Z, S, V, and H Flags are cleared.

Example: If Register 34H contains 2EH, the C Flag is set, and Register 12H contains 1BH, the statement:

SBC 34H, 12H
OpCode: 34 12 34

leaves the value 13H in Register 34H. The D Flag is set, and the C, Z, S, V, and H Flags are cleared.

Example: If Register 4BH contains 82H, the C Flag is set, Working Register R3 contains 10H, and Register 10H contains 01H, the statement:

SBC 4BH, @R3
OpCode: 35 E3 4B

leaves the value 80H in Register 4BH. The D Flag is set, and the C, Z, S, V, and H Flags are cleared.

Example: If Register 6CH contains 2AH, and the C Flag is not set, the statement:

SBC 6CH, #03H
OpCode: 36 6C 03

leaves the value 27H in Register 6CH. The D Flag is set, and the C, Z, S, V, and H Flags are cleared.

Example: If Register D4H contains 5FH, Register 5FH contains 4CH, and the C Flag is set, the statement:

SBC @D4H, #02H
OpCode: 37 D4 02

leaves the value 4AH in Register 5FH. The D Flag is set, and the C, Z, S, V, and H Flags are cleared.

SCF
SET CARRY FLAG

SCF
Set Carry Flag

SRC

Instruction Format:

	Cycles	OPC (Hex)
OPC	6	DF

Operation: C <— 1

The C Flag is set to 1, regardless of its previous value.

Flags:

C:	Set to 1
Z:	Unaffected
S:	Unaffected
V:	Unaffected
D:	Unaffected
H:	Unaffected

Example: If the C Flag is currently reset, the statement:

SCF
OpCode: DF

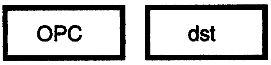
sets the Carry Flag to 1.

SRA
SHIFT RIGHT ARITHMETIC

SRA
Shift Right Arithmetic

SRA dst

Instruction Format:

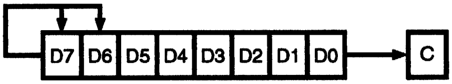


Cycles	OPC (Hex)	Address Mode dst
6	D0	R
6	D1	IR

Operation:

C <— dst(0)
dst(0) <— dst(1)
dst(1) <— dst(2)
dst(2) <— dst(3)
dst(3) <— dst(4)
dst(4) <— dst(5)
dst(5) <— dst(6)
dst(6) <— dst(7)
dst(7) <— dst(7)

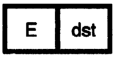
An arithmetic shift right by one bit position is performed on the destination operand. Bit 0 replaces the C Flag. Bit 7 (the Sign bit) is unchanged and its value is shifted into bit 6.



Flags:

C: Set if the bit rotated from the least significant bit position was 1 (i.e., bit 0 was 1).
Z: Set if the result is zero; cleared otherwise.
S: Set if the result bit 7 is set; cleared otherwise.
V: Always reset to 0.
D: Unaffected
H: Unaffected

Note: Address modes R or IR can be used to specify a 4-bit Working Register. In this format, destination Working Register operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.



SRA
SHIFT RIGHT ARITHMETIC

Example: If the contents of Working Register R6 are 31H (00110001B), the statement:

SRA R6
OpCode: D0 E6

leaves the value 98H (00011000) in Working Register R6. The C Flag is set, and the Z, V, and S Flags are cleared.

Example: If Register C6 contains the value DFH, and Register DFH contains the value B8H (10111000B), the statement:

SRA @C6
OpCode: D1 C6

leaves the value DCH (11011100B) in Register DFH. The C, Z, and V Flags are reset, and the S Flag is set.

SRP
SET REGISTER POINTER

SRP
Set Register Pointer

SRP src

Instruction Format:

	Cycles	OPC (Hex)	Address Mode dst
OPC | src	6	31	IM

Operation: RP <— src

The specified value is loaded into the Register Pointer (RP) (Control Register FDH). Bits 7-4 determine the Working Register Group within the Z8 Standard Register File. These Working Registers are selected when bits 3-0 are set to 0000B. When bits 3-0 are defined, the Expanded Working Register Bank is specified. The contents of bits 7-4 are disregarded when bits 3-0 are defined other than 0000B.

Register Pointer (FDH) Contents (Bin)	Working Register Group (Hex)	Actual Registers (Hex)
1111 0000	F	F0-FF
1110 0000	E	E0-EF
1101 0000	D	D0-DF
1100 0000	C	C0-CF
1011 0000	B	B0-BF
1010 0000	A	A0-AF
1001 0000	9	90-9F
1000 0000	8	80-8F
0111 0000	7	70-7F
0110 0000	6	60-6F
0101 0000	5	50-5F
0100 0000	4	40-4F
0011 0000	3	30-3F
0010 0000	2	20-2F
0001 0000	1	10-1F
0000 0000	0	00-0F

SRP
SET REGISTER POINTER

Register Pointer (FDH) Contents (Hex)	Working Register Group (Hex)	Working Registers (Dec)
xxxx 1111	F	R0-R15
xxxx 1110	E	R0-R15
xxxx 1101	D	R0-R15
xxxx 1100	C	R0-R15
xxxx 1011	B	R0-R15
xxxx 1010	A	R0-R15
xxxx 1001	9	R0-R15
xxxx 1000	8	R0-R15
xxxx 0111	7	R0-R15
xxxx 0110	6	R0-R15
xxxx 0101	5	R0-R15
xxxx 0100	4	R0-R15
xxxx 0011	3	R0-R15
xxxx 0010	2	R0-R15
xxxx 0001	1	R0-R15

Flags:

C: Unaffected
Z: Unaffected
S: Unaffected
V: Unaffected
D: Unaffected
H: Unaffected

Note: When an Expanded Register Bank is defined as the current Working Register, access to the Z8 Standard Register File is possible through direct addressing.

Example: The statement:

SRP F0H
OpCode: 70 F0

sets the Register Pointer to access Working Register Group F in the Z8 Standard Register File. All references to Working Registers now affect this group of 16 registers. Registers F0H to FFH can be accessed as Working Registers R0 to R15

SRP
SET REGISTER POINTER

Example: The statement:

SRP 0FH
OpCode: 70 0F

sets the Register Pointer to access Expanded Register Bank F as the current Working Registers. All references to Working Registers now affect this group of 16 registers. These registers are now accessed as Working Registers R0 to R15.

Example: Assume the RP currently addresses the Control and Peripheral Working Register Group and the program has just entered an interrupt service routine. The statement:

SRP 70H
OpCode: 31 70

retains the contents of the Control and Peripheral Registers by setting the RP to 70H (01110000B). Any reference to Working Registers in the interrupt routine will point to registers 70H to 7FH.

STOP
Stop

STOP

Instruction Format:

	Cycles	OPC (Hex)
OPC	6	6F

Operation: This instruction turns off the internal system clock (SCLK) and external crystal (XTAL) oscillation, and reduces the standby current. The STOP mode is terminated by a RESET which causes the processor to restart the application program at address 000CH.

Flags:

C:	Unaffected
Z:	Unaffected
S:	Unaffected
V:	Unaffected
D:	Unaffected
H:	Unaffected

Note: In order to enter STOP mode, it is necessary to first flush the instruction pipeline to avoid suspending execution in mid-instruction. The user must execute a NOP immediately before the execution of the STOP instruction.

Example: The statements:

NOP
STOP
OpCodes: FF 6F

place the Z8 into STOP mode.

SUB
SUBTRACT

SUB
Subtract

SUB dst, src

Instruction Format:

Format			Cycles	OPC (Hex)	Address dst	Mode src
OPC	dst | src		6	22	r	r
			6	23	r	Ir
OPC	src	dst	10	24	R	R
			10	25	R	IR
OPC	dst	src	10	26	R	IM
			10	27	IR	IM

Operation: dst <— dst - src

The source operand is subtracted from the destination operand and the result is stored in the destination operand. The contents of the source operand are not affected. Subtraction is performed by adding the two's complement of the source operand to the destination operand.

Flags:

C: Cleared if there is a carry from the most significant bit of the result; set otherwise, indicating a "borrow."
Z: Set if the result is 0; cleared otherwise.
V: Set if arithmetic overflow occurred (if the operands were of opposite sign and the sign of the result is the same as the sign of the source); reset otherwise.
S: Set if the result is negative; cleared otherwise.
H: Cleared if there is a carry from the most significant bit of the low order four bits of the result; set otherwise indicating a "borrow."
D: Always set to 1.

Note: Address modes R or IR can be used to specify a 4-bit Working Register. In this format, the source or destination Working Register operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.

E	src	or	E	dst

Example: If Working Register R3 contains 16H, and Working Register R11 contains 20H, the statement:

SUB R3, R11
OpCode: 22 3B

leaves the value F6H in Working Register R3. The C, S, and D Flags are set, and the Z, V, and H Flags are cleared.

SUB
SUBTRACT

Example: If Working Register R15 contains 16H, Working Register R10 contains 20H, and Register 20H contains 11H, the statement:

SUB R16, @R10
OpCode: 23 FA

leaves the value 05H in Working Register R15. The D Flag is set, and the C, Z, S, V, and H Flags are cleared.

Example: If Register 34H contains 2EH, and Register 12H contains 1BH, the statement:

SUB 34H, 12H
OpCode: 24 12 34

leaves the value 13H in Register 34H. The D Flag is set, and the C, Z, S, V, and H Flags are cleared.

Example: If Register 4BH contains 82H, Working Register R3 contains 10H, and Register 10H contains 01H, the statement:

SUB 4BH, @R3
OpCode: 25 E3 4B

leaves the value 81H in Register 4BH. The D Flag is set, and the C, Z, S, V, and H Flags are cleared.

Example: If Register 6CH contains 2AH, the statement:

SUB 6CH, #03H
OpCode: 26 6C 03

leaves the value 27H in Register 6CH. The D Flag is set, and the C, Z, S, V, and H Flags are cleared.

Example: If Register D4H contains 5FH, Register 5FH contains 4CH, the statement:

SUB @D4H, #02H
OpCode: 17 D4 02

leaves the value 4AH in Register 5FH. The D Flag is set, and the C, Z, S, V, and H Flags are cleared.

SWAP
SWAP NIBBLES

SWAP
Swap Nibbles

SWAP dst

Instruction Format:

OPC	dst

Cycles	OPC (Hex)	Address Mode dst
6	F0	R
6	F1	IR

Operation: dst(7-4) <—> dst(3-0)

The contents of the lower four bits and upper four bits of the destination operand are swapped.

Flags:

C: Unaffected
Z: Set if the result is zero; cleared otherwise.
S: Set if the result bit 7 is set; cleared otherwise.
V: Undefined
D: Unaffected
H: Unaffected

Note: Address modes R or IR can be used to specify a 4-bit Working Register. In this format, destination Working Register operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.

E	dst

Example: If Register BCH contains B3H (10110011B), the statement:

SWAP B3H
OpCode: F0 B3

will leave the value 3BH (00111011B) in Register BCH. The Z and S Flags are cleared.

Example: If Working Register R5 contains BCH and Register BCH contains B3H (10110011B), the statement:

SWAP @R5H
OpCode: F1 E5

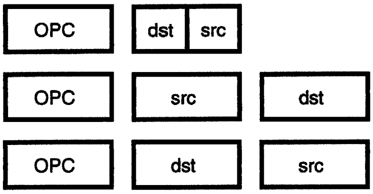
will leave the value 3BH (00111011B) in Register BCH. The Z and S Flags are cleared.

TCM
TEST COMPLEMENT UNDER MASK

TCM
Test Complement Under Mask

TCM dst, src

Instruction Format:



Cycles	OPC (Hex)	Address dst	Mode src
6	62	r	r
6	63	r	Ir
10	64	R	R
10	65	R	IR
10	66	R	IM
10	67	IR	IM

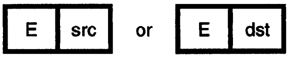
Operation: (NOT dst) AND src

This instruction tests selected bits in the destination operand for a logical 1 value. The bits to be tested are specified by setting a 1 bit in the corresponding bit position in the source operand (the mask). The TCM instruction complements the destination operand, and then ANDs it with the source mask (operand). The Zero (Z) Flag can then be checked to determine the result. If the Z Flag is set, then the tested bits were 1. When the TCM operation is complete, the destination and source operands still contain their original values.

Flags:

C:	Unaffected
Z:	Set if the result is zero; cleared otherwise.
S:	Set if the result bit 7 is set; cleared otherwise.
V:	Always reset to 0.
D:	Unaffected
H:	Unaffected

Note: Address modes R or IR can be used to specify a 4-bit Working Register. In this format, the source or destination Working Register operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.



Example: If Working Register R3 contains 45H (01000101B) and Working Register R7 contains the value 01H (00000001B) (bit 0 is being tested if it is 1), the statement:

TCM R3, R7
OpCode: 62 37

will set the Z Flag indicating bit 0 in the destination operand is 1. The V and S Flags are cleared.

TCM
TEST COMPLEMENT UNDER MASK

Example: If Working Register R14 contains the value F3H (11110011B), Working Register R5 contains CBH, and Register CBH contains 88H (10001000B) (bit 7 and bit 4 are being tested if they are 1), the statement:

TCM R14, @R5
OpCode: 63 E5

will reset the Z Flag, because bit 4 in the destination operand is not a 1. The V and S Flags are also cleared.

Example: If Register D4H contains the value 04H (000001000B), and Working Register R0 contains the value 80H (10000000B) (bit 7 is being tested if it is 1), the statement:

TCM D4H, R0
OpCode: 64 E0 D4

will reset the Z Flag, because bit 7 in the destination operand is not a 1. The S Flag will be set, and the V Flag will be cleared.

Example: If Register DFH contains the value FFH (11111111B), Register 07H contains the value 1FH, and Register 1FH contains the value BDH (10111101B) (bit 7, bit 5, bit 4, bit 3, bit 2, and bit 0 are being tested if they are 1), the statement:

TCM DFH, @07H
OpCode: 65 07 DF

will set the Z Flag indicating the tested bits in the destination operand are 1. The S and V Flags are cleared.

Example: If Working Register R13 contains the value F1H (11110001B), the statement:

TCM R13, #02H
OpCode: 66 ED, 02

tests bit 1 of the destination operand for 1. The Z Flag will be set indicating bit 1 in the destination operand was 1. The S and V Flags are cleared.

Example: If Register 5DH contains A0H, and Register A0H contains 0FH (00001111B), the statement:

TCM 5D, #10H
OpCode: 67 5D 10

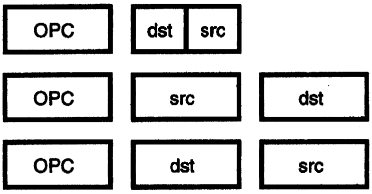
tests bit 4 of the Register A0H for 1. The Z Flag will be reset indicating bit 1 in the destination operand was not 1. The S and V Flags are cleared.

TM
TEST UNDER MASK

TM
Test Under Mask

TM dst, src

Instruction Format:



Cycles	OPC (Hex)	Address dst	Mode src
6	72	r	r
6	73	r	Ir
10	74	R	R
10	75	R	IR
10	76	R	IM
10	77	IR	IM

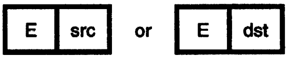
Operation: dst AND src

This instruction tests selected bits in the destination operand for logical a 0 value. The bits to be tested are specified by setting a 1 bit in the corresponding bit position in the source operand (the mask). The TCM instruction ANDs the destination operand with the source operand (the mask). The Zero (Z) Flag can then be checked to determine the result. If the Z Flag is set, then the tested bits were 0. When the TCM operation is complete, the destination and source operands still contain their original values.

Flags:

C:	Unaffected
Z:	Set if the result is zero; cleared otherwise.
S:	Set if the result bit 7 is set; cleared otherwise.
V:	Always reset to 0.
D:	Unaffected
H:	Unaffected

Note: Address modes R or IR can be used to specify a 4-bit Working Register. In this format, the source or destination Working Register operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.



Example: If Working Register R3 contains 45H (01000101B) and Working Register R7 contains the value 01H (00000010B) (bit 1 is being tested if it is 0), the statement:

TM R3, R7
OpCode: 72 37

will set the Z Flag indicating bit 1 in the destination operand is 0. The V and S Flags are cleared.

TM
TEST UNDER MASK

Example: If Working Register R14 contains the value F3H (11110011B), Working Register R5 contains CBH, and Register CBH contains 88H (10001000B) (bit 7 and bit 4 are being tested if they are 0), the statement:

TM R14, @R5
OpCode: 73 E5

will reset the Z Flag, because bit 4 in the destination operand is not a 0. The S Flag will be set, and the V Flag is cleared.

Example: If Register D4H contains the value 08H (00001000B), and Working Register R0 contains the value 04H (00001000B) (bit 3 is being tested if it is 0), the statement:

TM D4H, R0
OpCode: 74 E0 D4

will set the Z Flag, because bit 3 in the destination operand is a 0. The S and V Flags will be cleared.

Example: If Register DFH contains the value 00H (00000000B), Register 07H contains the value 1FH, and Register 1FH contains the value BDH (10111101B) (bit 7, bit 5, bit 4, bit 3, bit 2, and bit 0 are being tested if they are 0), the statement:

TM DFH, @07H
OpCode: 75 07 DF

will set the Z Flag indicating the tested bits in the destination operand are 0. The S is set, and the V Flag is cleared.

Example: If Working Register R13 contains the value F1H (11110001B), the statement:

TM R13, #02H
OpCode: 76 ED, 02

tests bit 1 of the destination operand for 0. The Z Flag will be set indicating bit 1 in the destination operand was 0. The S and V Flags are cleared.

Example: If Register 5DH contains A0H, and Register A0H contains 0FH (00001111B), the statement:

TM 5D, #10H
OpCode: 77 5D 10

tests bit 4 of the Register A0H for 0. The Z Flag will be set indicating bit 1 in the destination operand was 0. The S and V Flags are cleared.

WDH
WATCH-DOG TIMER ENABLE DURING HALT MODE

WDH
Watch-Dog Timer Enable During HALT Mode

WDH

Instruction Format:

	Cycles	OPC (Hex)
OPC	6	4F

Operation: When this instruction is executed it will enable the WDT (Watch-Dog Timer) during HALT mode. If this instruction is not executed the WDT will stop when entering HALT mode. This instruction does not clear the counter, it just makes it possible to have the WDT function running during HALT mode. A WDH instruction executed without executing WDT (5FH) has no effect.

Flags:

C:	Unaffected
Z:	Undefined
S:	Undefined
V:	Undefined
D:	Unaffected
H:	Unaffected

Note: The WDH instruction should not be used following any instruction in which the condition of the flags is important.

Example: If the WDT is enabled, the statement:

WDH
OpCode: 4F

will enable the WDT in HALT mode.

Note: This instruction format is valid only for the Z86C04/C07/C08 and Z86E04/E07/E08.

WDT
WATCH-DOG TIMER

WDT
Watch-Dog Timer

WDT

Instruction Format:

	Cycles	OPC (Hex)
OPC	6	5F

Operation: The WDT (Watch-Dog Timer) is a retriggerable one shot timer that will reset the Z8 if it reaches its terminal count. The WDT is initially enabled by executing the WDT instruction. Each subsequent execution of the WDT instruction refreshes the timer and prevents the WDT from timing out.

Flags:

C:	Unaffected
Z:	Undefined
S:	Undefined
V:	Undefined
D:	Unaffected
H:	Unaffected

Note: The WDT instruction should not be used following any instruction in which the condition of the flags is important.

Example: If the WDT is enabled, the statement:

WDT
OpCode: 5F

refreshes the Watch-Dog Timer.

Example: The first execution of the statement:

WDT
OpCode: 5F

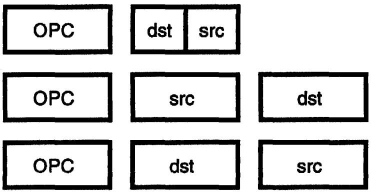
enables the Watch-Dog Timer.

XOR
LOGICAL EXCLUSIVE OR

XOR
Logical Exclusive OR

XOR dst, src

Instruction Format:



	OPC	Address	Mode
Cycles	(Hex)	dst	src
6	B2	r	r
6	B3	r	Ir
10	B4	R	R
10	B5	R	IR
10	B6	R	IM
10	B7	IR	IM

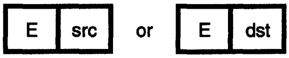
Operation: dst <— dst XOR src

The source operand is logically EXCLUSIVE ORed with the destination operand. The XOR operation results in a 1 being stored in the destination operand whenever the corresponding bits in the two operands are different, otherwise a 0 is stored. The contents of the source operand are not affected.

Flags:

C:	Unaffected
Z:	Set if the result is zero; cleared otherwise.
S:	Set if the result of bit 7 is set; cleared otherwise.
V:	Always reset to 0
D:	Unaffected
H:	Unaffected

Note: Address modes R or IR can be used to specify a 4-bit Working Register. In this format, the source or destination Working Register operand is specified by adding 1110B (EH) to the high nibble of the operand. For example, if Working Register R12 (CH) is the destination operand, then ECH will be used as the destination operand in the OpCode.



Example: If Working Register R1 contains 34H (00111000B) and Working Register R14 contains 4DH (10001101B), the statement:

XOR R1, R14
OpCode: B2 1E

leaves the value BDH (10111101B) in Working Register R1. The Z, and V Flags are cleared, and the S Flag is set.

XOR
LOGICAL EXCLUSIVE OR

Example: If Working Register R4 contains F9H (11111001B), Working Register R13 contains 7BH, and Register 7B contains 6AH (01101010B), the statement:

XOR R4, @R13
OpCode: B3 4D

leaves the value 93H (10010011B) in Working Register R4. The S Flag is set, and the Z, and V Flags are cleared.

Example: If Register 3AH contains the value F5H (11110101B) and Register 42H contains the value 0AH (00001010B), the statement:

XOR 3AH, 42H
OpCode: B4 42 3A

leaves the value FFH (11111111B) in Register 3AH. The S Flag is set, and the C and V Flags are cleared.

Example: If Working Register R5 contains F0H (11110000B), Register 45H contains 3AH, and Register 3A contains 7F (01111111B), the statement:

XOR R5, @45H
OpCode: B5 45 E5

leaves the value 8FH (10001111B) in Working Register R5. The S Flag is set, and the C and V Flags are cleared.

Example: If Register 7AH contains the value F7H (11110111B), the statement:

XOR 7AH, #F0H
OpCode: B6 7A F0

leaves the value 07H (00000111B) in Register 7AH. The Z, V and S Flags are cleared.

Example: If Working Register R3 contains the value 3EH and Register 3EH contains the value 6CH (01101100B), the statement:

XOR @R3, #05H
OpCode: B7 E3 05

leaves the value 69H (01101001B) in Register 3EH. The Z, V, and S Flags are cleared.

USER'S MANUAL

CHAPTER 13

ZILOG EMULATORS/SOFTWARE

13.1 ZILOG Z8 EMULATOR PRODUCTS

Zilog provides a family of full-featured real-time in-circuit emulators to support Z8® product development. In-circuit emulation links your design to a PC to determine how the microcontroller is functioning in your design. This greatly simplifies system debug, reducing development time and OTP device consumption. All emulators include OTP programming, a user configurable WINDOWS interface, a Zilog Z8® cross assembler and complete Z8® documentation. Product specifications for the following in-circuit emulator kits are also provided:

- Z86CCP00ZEM / Z86CCP00ZAC
- Z86C1200ZEM
- Z86C5000ZEM

13.2 Z8® CCP™ EMULATOR

QUICK START

① Check Support Package Contents

(See Other Side)

② Load Software

1. Select the "Run" command from the "File" menu, located under Microsoft Windows "Program Manager".
   a. Insert the disk labeled "Zilog ZASM Cross Assembler/Zilog MOBJ Object File Util." in drive A (or drive B, if appropriate.)
   b. Type "a:\setup" and press ENTER. (Type "b:\setup" if drive B is used.)
   c. Follow on-screen instructions.
   d. Remove diskette and store in a safe place when done.

For more information on assembling source code, refer to Z8 CCP Emulator User's Guide (Appendix C) and the Z8® Microcontrollers Technical Manual.

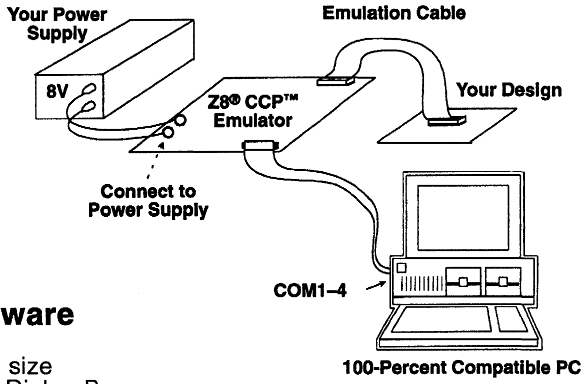
2. Select the "Run" command from the "File" menu, located under Microsoft Windows "Program Manager".
   a. Insert disk labeled "Z8 GUI S/W" in drive A (or drive B, if appropriate.)
   b. Type "a:\setup" and press ENTER. (Type "b:\setup" if drive B is used.)
   c. Follow on-screen instructions.
   d. Remove diskette and store in a safe place when done.

③ Make Connections

Power Supply, PC, and Your Design

Refer to Z8® CCP™ Emulator User's Manual

Observe Electrical Safeguards
(See Z8 CCP Emulator User's Manual)



④ Run Zilog ICEBOX GUI Software

1. Double click the Z8-ICE icon.
2. Select the microcontroller and ROM size to be emulated in the Configuration Dialog Box.
3. Use the "File" menu to download sample files to Z8 Code Memory.
4. Refer to Z8 CCP Emulator User's Manual, "Chapter 3: Z8 Emulator Sample Session".

13.3 Z8® CCP™ EMULATOR

PACKAGE CONTENTS

SUPPORT PRODUCTS PACKAGE CONTENTS

The Zilog Z8® CCP™ Emulator Support Products Package contains the following items:

Hardware

Z8® CCP™ Emulator Board
18-Pin DIP-to-DIP Target Cable
Z86E08 18-Pin DIP OTP Device

Software

Z8® GUI S/W Diskette
Zilog ZASM Cross Assembler/MOBJ Object File Util. Diskette
Production Languages Corporation (PLC) Compass/Z8™ Diskette (Evaluation Version)

Description of Z8® GUI Diskette Include Files

z8cfg.o	Configuration
z8ice.exe	Executable
icehelp.hlp	Help
meter.dll	Installation library
readme	Text file
setup.inf	Installation information
setup.exe	Windows install program
z8em_c12.o	On board software for Z86C12 Icebox
z8em_c27.o	On board software for Z86C27 Icebox
z8em_c50.o	On board software for Z86C50 Emulator
z8em_c62.o	On board software for Z86C62 Emulator
z8em_c65.o	On board software for Z89C65 Emulator
z8em_c67.o	On board software for Z89C67 Emulator
z8em_c93.o	On board software for Z86C93 Emulator
z8em_l7x.o	On board software for Z86L7X Emulator
z8em_ccp.o	On board software for Z86CCP Emulator

Publications

Zilog Z8 CCP Emulator User's Manual
Z8 Microcontrollers Technical Manual
Discrete Z8 Microcontrollers Databook
Registration Card

Optional Accessory Kit

An optional accessory kit (P/N Z86CCP00ZAC) available from Zilog contains the following items:
28-Pin ZIF Socket
40-Pin ZIF Socket
Power Cable
28-Pin DIP-to-DIP Target Cable
40-Pin DIP-to-DIP Target Cable

13. 4 Z86CCP00ZEM EMULATOR

PRODUCT SPECIFICATION

DEVICES SUPPORTED: Z86C03, Z86C04/E04, Z86C06, Z86C08/E08, Z86C09/19, Z86E03/E06; WITH Z8® CCP™ EMULATOR ACC. KIT (Z86CCPZAC): Z86C30/E30, Z86C31/E31, Z86C40/E40, Z86730, Z86C32

DESCRIPTION

The Z86CCP00ZEM is a member of Zilog's family of in-circuit emulators. The Z8 CCP emulator provides emulation and OTP programming support for Zilog's Consumer Controller Processor (CCP™) microcontroller. The Emulator provides all the essential MCU timing and I/O circuitry which simplifies user emulation of the prototype hardware/software product.

The data entering, program debugging, and OTP programming are performed by the monitor ROM and the Host Package which communicates through RS-232C serial interface with a fixed 19200 baud rate. The user program can be downloaded directly from the host computer via an RS-232C connector. The user code may then be executed using various debugging commands in the monitor. The Emulator can be connected to a serial port (COM 1, COM 2, COM3, COM4) of the host computer (386 or 486, IBM compatible PC) and uses Graphical User Interface (GUI) software.

SPECIFICATIONS

Emulation Specification

Maximum Emulation Speed: 8 MHz
Minimum Emulation Speed: 1 MHz

Power Requirements

+8V Vdc @ 0.5 A

Dimensions

Width:	7.0 in. (17.7 cm)
Length:	9.0 in. (22.9 cm)
Height:	0.9 in. (2.3 cm)

Serial Interface

RS-232C @ 19200 baud

KIT CONTENTS

Z8® CCP™ Emulator
CMOS Z86C9320VSC
RS-232C Interface
Reset Switch
20 MHz CMOS Z86C5020FSE ICE Chip
8K x 8 STATIC RAM (for Code Memory)
18-Pin DIP ZIF Programming Socket
18-Pin Target Connector Cable
Holes Available for 28/40-Pin ZIF Sockets
Sockets Available for 18/28/40-Pin Target Cables

Software (IBM PC Platform)

ZASM Cross-Assembler and MOBJ Object File Util.
Z8® GUI Emulator Software
Production Languages Corporation COMPASS/Z8
(Evaluation Version)

System Requirements

386 or 486, IBM Compatible PC
VGA Video Adapter (Color Monitor Recommended)
20 MHz, Minimum
4 Mbytes RAM
Microsoft Windows 3.0 or 3.1
Hard Disk Drive (1 Mbyte Free Space)
High Density (HD) Floppy Disk Drive (3.5-Inch)
RS-232 COM Port

Documentation

Registration Card
Product Information
Z8® CCP™ Emulator User's Manual
Discrete Z8 Databook
Z8® Microcontroller User's Manual

ORDERING INFORMATION

Part No: Z86CCP00ZEM

13.5 Z86CCP00ZAC EMULATOR KIT

PRODUCT SPECIFICATION

DESCRIPTION

The Z86CCP00ZAC is the accessory kit for the Z86CCP00ZEM. The kit contains all accessories to fully populate and operate all functions of the Z86CCP00ZEM.

KIT CONTENTS

Z8 CCP Emulator Accessory Kit

28-Pin ZIF Socket
28-Pin Target Connector Cable
40-Pin ZIF Socket
40-Pin Target Connector Cable
RS-232 Cable
Power Cable

ORDERING INFORMATION

Part No: Z86CCP00ZAC

13.6 Z86C1200ZEM EMULATOR

PRODUCT SPECIFICATION

DEVICES SUPPORTED: Z86117/717, Z86C04/E04, Z86C07/E07, Z86C08/E08, Z86C11, Z86C20, Z86C21/E21, Z86E22, Z86E23, Z86C60, Z86C61/E61, Z86C63/E63, Z86C65, Z86C91

DESCRIPTION

The Z86C1200ZEM Z8® Emulator is a member of Zilog's ICEBOX™ product family of in-circuit emulators. The Z86C1200ZEM provides emulation and OTP programming support for Zilog's Z8 microcontrollers. The Emulator provides all the essential MCU timing and I/O circuitry which simplifies user emulation of the prototype hardware/ software product. The data entering, program debugging, and OTP programming are performed by the monitor ROM and the Host Package which communicates through a RS-232C serial interface with a fixed 19200 baud rate. The user program can be downloaded directly from the host computer through the RS-232C connector. The user code may then be executed using various debugging commands in the monitor. The Emulator can be connected to a serial port (COM 1, COM 2, COM3, COM4) of the host computer (386 or 486, IBM compatible PC) and uses Graphical User Interface (GUI) software.

SPECIFICATIONS

Emulation Specification

Minimum Emulation Speed: 1 MHz
Maximum Emulation Speed: 16 MHz

Power Requirements

+5 Vdc @ 0.5 A

Dimensions

Width: 6.25 in. (15.8 cm)
Length: 9.5 in. (24.1 cm)
Height: 2.5 in. (6.35 cm)

Serial Interface

RS-232C @ 19200 baud

KIT CONTENTS

Z86C12 Emulator

Z8® Emulation Base Board

CMOS Z86C9120PSC
8K X 8 EPROM (Programmed with Debug Monitor)
- 32K X 8 STATIC RAM
- 3 64K X 4 STATIC RAM
- RS-232C Interface
- Reset Switch

Z86C12 Emulation Daughter Board
- 16 MHz CMOS Z86C1216GSE ICE Chip
- 18/40-Pin ZIF OTP Sockets
- 40/60/80-Pin Target Connectors

Cables/Pods

18-Pin DIP Emulation Cable
28-Pin DIP Emulation Cable
40-Pin DIP Emulation Cable
Power Cable with Banana Plugs
Power Cable with 1A Slow-Blow Fuse
DB 25 RS-232C Cable

Software (IBM®-PC Platform)

ZASM Cross-Assembler and MOBJ Object File Util.
Z8® GUI Emulator Software

Documentation

Emulator User's Manual
Z8® Cross-Assembler User's Guide
Universal Object File Utilities (MOBJ) User's Guide
Registration Card
Product Information

ORDERING INFORMATION

Part No: Z86C1200ZEM

13.7 Z86C5000ZEM EMULATOR

PRODUCT SPECIFICATION

DEVICES SUPPORTED: Z86C03, Z86C06, Z86C09/19, Z86C30/E30, Z86C31/E31, Z86C40/E40, Z86C89, Z86C90, Z86L06, Z86L29, Z86E03/E06, Z86C32, Z86730

DESCRIPTION

The Z86C5000ZEM (C50) Emulator is a member of Zilog's ICEBOX™ product family of in-circuit emulators. The C50 Emulator provides emulation and OTP programming support for Zilog's CCP™ (Consumer Controller Processor) microcontrollers. The C50 Emulator provides all the essential MCU timing and I/O circuitry which simplifies user emulation of the prototype hardware/software product. The Emulator can be connected to a serial port (COM 1, COM 2, COM3, COM4) of the host computer (386 or 486, IBM compatible PC) and uses Graphical User Interface (GUI) software.

SPECIFICATIONS

Emulation Specification

Miinimum Emulation Speed: 1 MHz
Maximum Emulation Speed: 20 MHz

Power Requirements

+5V Vdc @ 1.0 A

Dimensions

Width: 6.25 in. (15.8 cm)
Length: 9.5 in. (24.1 cm)
Height: 2.5 in. (6.35 cm)

Serial Interface

RS-232C @ 19200 baud

System Requirements

386 or 486, IBM Compatible PC
VGA Video Adapter (Color Monitor Recommended)
20 MHz, Minimum
4 Mbytes RAM
Microsoft Windows 3.0 or 3.1
Hard Disk Drive (1 Mbyte Free Space)
High Density (HD) Floppy Disk Drive (3.5-Inch)
RS-232 COM Port

KIT CONTENTS

Z86C50 Emulator

Z8® Emulation Base Board
- CMOS Z86C9120PSC
- 8K x 8 EPROM (Programmed with Debug Mtr.)
- 32K x 8 Static RAM
- 3 64K x4 Static RAMs
- RS-232C Interface
- Reset Switch

Z86C50 Emulation Daughter Board
- 20 MHz CMOS Z86C5020GSE ICE Chip
- 2K x 8 Static RAM
- 18/28/40-Pin ZIF OTP Sockets
- 6 HP-16500A Logic Analysis System Interface Connectors
- 40/60/80-Pin Target Connectors

Cables

40-Pin DIP Emulation Cable
28-Pin DIP Emulation Cable
18-Pin DIP Emulation Cable
Power Cable with Banana Plugs
DB25 RS-232C Cable

Software (IBM PC Platform)

ZASM Cross-Assembler and MOBJ Object File Util.
Z8® GUI Emulator Software

Documentation

ICEBOX™ User's Manual
Z8 Cross-Assembler User's Guide
Windows Host Interface User's Guide (GUI)
Universal Object File Utilities (MOBJ) User's Guide
Registration Card

ORDERING INFORMATION

Part No Z86C5000ZEM

13.8 SOFTWARE

13.8.1 INTRODUCTION

This section describes some of the important features of the Z8®, with software examples that illustrate its power and ease of use. It is divided into sections by topic; the user need not read each section sequentially, but may skip around to the sections of current interest.

For feature availability and implementation details on a particular Z8 device, see the product specification.

13.9 ACCESSING REGISTER MEMORY

The Z8 register space consists of I/O ports, control and status registers, and general-purpose registers. The general-purpose registers are RAM areas typically used for accumulators, pointers, and stack area. This section describes these registers and how they are used. Bit manipulation and stack operations effecting the register space are discussed in other sections of this manual.

13.9.1 Registers and Register Pairs

The Z8 supports 8-bit registers and 16-bit register pairs. A register pair consists of a an even-numbered register concatenated with the next higher numbered register (00 and 01, 02 and 03, ... FFH). A register pair must be addressed by reference to the even-numbered register.

- F1H and F2H are not a valid register pairs.
- F0H and F1H are valid register pairs, addressed by reference to F0H.

Register pairs may be incremented (INCW) and decremented (DECW) and are useful as pointers for accessing program and external data memory.

Any instruction which can reference or modify an 8-bit register can do so to any of the registers in the Z8, regardless of the inherent nature of that register. Thus, I/O ports, control, status, and general-purpose registers may all be accessed and manipulated without the need for special-purpose instructions. Similarly, instructions which reference or modify a 16-bit register pair can do so to any of the valid register pairs.

The only exceptions to this rule are as follows:

- The DJNZ (decrement and jump if non-zero) instruction may successfully operate on the general-purpose working registers only.
- All write-only control registers may be modified only by such instructions as LOAD, POP, and CLEAR. Instructions such as OR and AND require that the current contents of the operand be readable and therefore will not function properly on the write-only registers.

13.9.2 Register Pointer

Within the register addressing modes provided by the Z8®, a register may be specified by its full 8-bit address (00H-FFH) or by a short 4-bit address. In the latter case, the register is viewed as one of the 16 working registers within a working register group. Such a group must be aligned on a 16-byte boundary and is addressed by Register Pointer RP (FDH). As an example, assume the Register Pointer contains 70, thus pointing to the working register group from 70H to 7FH. The LD instruction may be used to initialize register 76H to an immediate value in one of two ways

```
LD 76,#01H     !8-bit register address is given
               by instruction (3 byte instruction)!

or

LD R6,#01H     !4-bit working register address is
               given by instruction; 4-bit work
               ing register group address is
               given by Register Pointer (2 byte
               instruction)!
```

The address calculation for the latter case is illustrated in Figure 13.1. Notice that 4-bit working-register addressing offers code compactness and fast execution compared to its 8-bit counterpart.

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

```
SRP    #value
```

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

```
LD     RP, #value
```

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

```
SRP    #70H
```

is used to set the Register Pointer for the previous example.

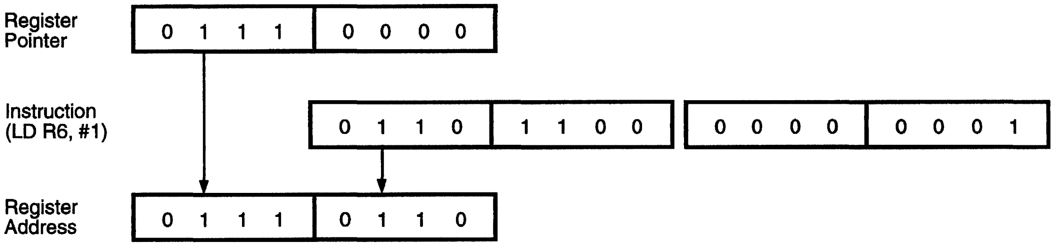


Figure 13-1. Address Calculation Using The Register Pointer

13.9.3 Context Switching

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

```
PUSH RP      !preserve Register Pointer of in
              terrupted task!
SRP #40H     !address working register group
              4!
```

Before exiting, the service routine should execute the instruction

```
POP    RP
```

to restore the Register Pointer to its entry value.

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

13.9.4 Addressing Mode

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

13.9.5 Direct Register Addressing

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

```
LD FEH,#HI STACK

            !load register FEH (SPH) with the upper 8-
            bits of the label STACK!

AND 00H,MASK_REG

            !AND register 0 with register named
            MASK_REG!

OR 01H,R5

            !OR register 1 with working register 5!
```

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

```
INCW   RRO

            !increment working register pair R0, R
            R1 = R1 + 1
            R0 = R0 + carry!

DECW 7EH

            !decrement working register pair 7EH,
            7FH
            7FH = 7FH - 1
            7EH = 7EH - carry!
```

Note that the instruction

```
INCW RR5
```

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

13.9.6 Indirect Register Addressing

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

```
            LD      R1,#20H

                    !working register 1 is the byte counter
                    output 20H bytes!

            LD      R2,#60H

                    !working register 2 is the buffer pointer
                    register!
out_again:

            LD      R0,@R2

                    !load into working register 0 the byte pointed
                    to by working register 2!

            INC     R2          !increment pointer!

            CALL  SERIAL_OUT

                    !output the byte!

            DJNZ    R1,out_again

                    !loop till done!
```

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

```
INCW   @R0

        !increment the register pair whose ad
        dress is contained in working register 0!

DECW @7FH

        !decrement the register pair whose
        address is contained in register 7FH!
```

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

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

```
SUB    4,BASE(R5)
```

requires the indexed addressing mode which is not directly supported by the Z8® subtract instruction. This instruction can be simulated as follows

```
LD     R6,#BASE

       !working register 6 has the base address!

ADD    R6,R5

       !calculate the target address!

SUB  04H,@R6

       !now use indirect addressing to perform
       the actual subtract!
```

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

13.9.7 Indexed Addressing

The indexed addressing mode is supported by the load instruction (LD) for the transference of bytes between a working register and another register. The effective address of the latter register is given by the instruction which is offset by the contents of a designated working (index) register. This addressing mode provides efficient memory usage when addressing consecutive bytes in a block of register memory, such as a table or a buffer. The working register used as the index in the effective address calculation can serve the additional role of counter to control a loop's duration.

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

```
         LD     R0,#LENGTH

                !length of buffer!
                !starting at buffer end, look for 1st non-
                blank!
loop:
         LD     R1,BUF-1(R0)
         CP     R1,#' '
         JR     ne,found

                !found non-blank!

         DJNZ   R0,loop

                !look at next!
all_blanks:     !length = 0!

found
         5 instructions
         12 bytes
         6 cycles overhead
         42 cycles per character tested
```

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

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

         !starting at buffer end, look for 1st non-blank!

loop1:

         CP     @R1,#' '
         JR     ne,found1

                !found non-blank!

         DEC R1

                !dec pointer!

         DJNZ   R0,loop1

                !are we done?!

all_blanks1:    !length = 0!

found1:

         6 instructions
         13 bytes
         12 cycles overhead
         38 cycles per character tested
```

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

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

```
        LD      R0,# - LENGTH

                !starting at buffer start, look for 1st car
                riage return (= 0DH)!
next:

        LD      r1,BUF + LENGTH(R0)
        CP      R1,#0DH
        JR      eq,cr

                !found it!

        INC     R0

                !update counter/index!

        JR      nz,next

                !try again!
cr:

        ADD     R0,#LENGTH

                !R0 has length to CR!
```

7 instructions
16 bytes
6 cycles overhead
48 cycles per character tested

13.10 Accessing Program and External Data Memory

In a single instruction, the Z8® can transfer a byte between register memory and either program or external data memory. Load Constant (LDC) and Load Constant and Increment (LDCI) reference program memory; Load External (LDE) and Load External and Increment (LDEI) reference external data memory. These instructions require that a working register pair contain the address of the byte in either Program or External Data Memory to be accessed by the instruction (indirect working register pair addressing mode). The register byte operand is specified by using the direct working register addressing mode in LDC and LDE or the indirect working register addressing mode in LDCI and LDE1. In addition to performing the designated byte transfer, LDCI and LDEI automatically increment both the indirect registers specified by the instruction. These instructions are therefore efficient for performing block moves between register and either program or external data memory. Since the indirect addressing mode is used to specify the operand address within program or external data memory, more complex addressing modes may be simulated. For example, the instruction

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

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

```
        LD      R0,#HI BASE
        LD      R1,#LO BASE

                !RRO has table start address!

        ADD     R1,R2
        ADC     R0,#0

                !RRO has table entry address!

        LDC     R3,@RR0

                !R3 has the table entry!
```

13.10.1 Configuring the Z8 for I/O Applications as Opposed to Memory Intensive Applications

The Z8 offers a high degree of flexibility in memory and I/O intensive applications. For devices with thirty-two port bits provided, 16, 12, eight, or zero may be configured as address bits to external memory. This allows for addressing of up to 64K bytes of external memory, which can be expanded to 128K bytes if the Data Memory Select output (DM) is used to distinguish between program and data memory accesses. The following instructions illustrate the code sequence required to configure the Z8 with 12 external addressing lines and to enable the Data Memory Select output:

```
        LD      PO1M,# 00010010B

                        !bit 3-4 enable AD0-AD7;
                        bit 0-1 enable A8-A11!

        LD      P3M,# 00010010B
                        !bit 3-4 enable DM!
```

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

13.10.2 LDC and LDE

To illustrate the use of the Load Constant (LDC) and Load External (LDE) instructions, assume there exists a hardware configuration with external memory and Data Memory Select enabled.

13.10.3 Accessing Program and External Data Memory

LDCI instruction provides an economical means of initializing consecutive registers from an initialization table in program memory. The following code excerpt illustrates this technique of initializing control registers F2H through FFH from a 14-byte array (INIT_tab) in program memory:

```
        SRP     #00H

        LD      R6,#HI INIT_tab
        LD      R7,#LO INIT_tab
        LD      R8,#F2H

                        !1st reg to be initialized!

        LD      R9,#0EH

                        !length of register block!
loop:
        LDCI    @R8,@RR6

                        !load a register from the init table!

        DJNZ    R9,loop

                        !continue till done!
```

7 instructions
14 bytes
30 cycles overhead
30 cycles per register initialized

13.10.4 LDEI

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

```
        LD      R10,#HI SAVE

                        !external memory!

        LD      R11,#LO SAVE

                        !address!

        LD      R8,#40H

                        !starting register!

        LD      R9,#21H         sponding mask bit is a
                                logic 1.
                        !number of registers to save in
                        external data memory!
loop:
        LDEI    @RR10,@R8

                        !init a register!

        DJNZ    R9,loop

                        !until done!
```

6 instructions
12 bytes
24 cycles overhead
30 cycles per register saved

13.11 BIT MANIPULATIONS

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

These functions are supported in the Z8 primarily by six instructions:

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

These instructions may access any Z8® register, regardless of its inherent type (control, I/O, or general-purpose), with the exception of the write-only control registers. Table 13-1 summarizes the function performed on the destination byte by each of the above instructions. All of these instructions, with the exception of COM, require a mask operand. The 'selected' bits referenced in Table 13-1 are those bits in the destination operand for which the corre-

Table 13-1 Bit Manipulation Instruction Usage

Opcode	Use
TM	To test selected bits for logic 0
TCM	To test selected bits for logic 1
AND	To reset all but selected bits to logic 0
OR	To set selected bits to logic 1
XOR	To complement selected bits
COM	To complement all bits

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

13.11.1 Test Under Mask (TM)

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

destination AND source.

Neither source nor destination operand is modified; the FLAGS control register is the only register affected by this instruction. The zero flag (Z) is set if all selected bits are logic 0; it is reset otherwise. Thus, if the selected destination bits are either all logic 1 or a combination of 1s and 0s, the zero flag would be cleared by this instruction. The sign flag (S) is either set or reset to reflect the result of the AND operation; the overflow flag (V) is always reset. All other flags are unaffected. Table 13-2 illustrates the flag settings which result from the TM instruction on a variety of source and destination operand combinations. Note that a given TM instruction will never result in both the Z and S flags being set.

13.11.2 Test Complement Under Mask

The Test Complement under Mask instruction is used to test selected bits for logic 1. The logical operation performedis

(NOTdestination)ANDsource.

Table 13-2 Effects of the TM Instruction

Destination	Source	Flags
(binary)	(binary)	Z S V
10001100	01110000	1 0 0
01111100	01110000	0 0 0
10001100	11110000	0 1 0
11111100	11110000	0 1 0
00011000	10100001	1 0 0
01000000	10100001	1 0 0

As in Test under Mask, the FLAGS control register is the only register affected by this operation. The zero flag (Z) is set if all selected destination bits are 1; it is reset otherwise. The sign flag (S) is set or reset to reflect the result of the AND operation; the overflow flag (V) is always reset. Table 13-3 illustrates the flag settings which result from the TCM instruction on a variety of source and destination operand combinations. As with the TM instruction, a given TCM instruction will never result in both the Z and S flags being set.

Table 13-3 Effects of the TCM Instruction

Destination	Source	Flags
(binary)	(binary)	Z S V
10001100	01110000	0 0 0
01111100	01110000	1 0 0
10001100	11110000	0 0 0
11111100	11110000	1 0 0
00011000	10100001	0 1 0
01000000	10100001	0 1 0

13.12 Stack Operations

The Z8® stack resides within an area of data memory (internal or external). The current address in the stack is contained in the stack pointer, which decrements as bytes are pushed onto the stack, and increments as bytes are popped from it. The stack pointer occupies two control register bytes (FEH and FFH) in the Z8 register space and may be manipulated like any other register. The stack is useful for subroutine calls, interrupt service routines, and parameter passing and saving. Figure 13-2 illustrates the downward growth of a stack as bytes are pushed onto it.

13.12.1 Internal as Opposed to External Stack

The location of the stack in data memory may be selected to be either internal register memory or external data memory. Bit 2 of control register P01M (F8H) controls this selection. Register pair SPH (FEH), SPL (FFH) serves as the stack pointer for an external stack. Register SPL is the stack pointer for an internal stack.

In the latter configuration, SPH is available for use as a general purpose register. The following illustrates a code sequence that initializes external stack operations:

```
LD    P01M,#00H

                    !bit 2: select external
                    stack!

LD    SPH,#HI       ;STACK

LD    SPL,#LO       ;STACK
```

13.12.2 CALL

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

```
LABEL 1     CALL  4F98H

            !direct addressing: PC is
            loaded with the hex value 4F98;
            address LABEL 1+3 is pushed
            onto the stack!

LABEL 2     CALL  @RR4

            !indirect addressing: PC is
            loaded with the contents of
            working register pair R4, R5;
            address LABEL 2+2 is pushed
            onto the stack!

LABEL 3     CALL  @7EH

            !indirect addressing PC is
            loaded with the contents of
            register pair 7EH, 7FH;
            address LABEL 3+2 is pushed
            onto the stack!
```

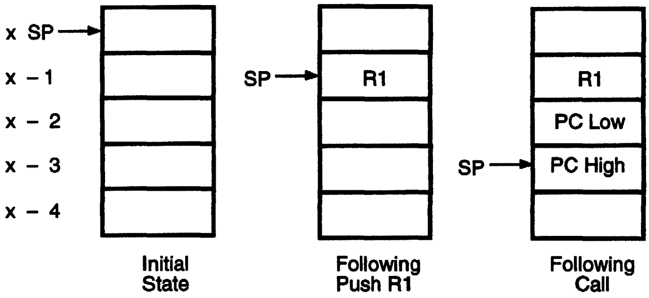


Figure 13-2. Growth Of A Stack

13.12.3 RET

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

13.12.4 Interrupt Machine Cycle

During an interrupt machine cycle, the PC followed by the status flags is pushed onto the stack. A more detailed discussion of interrupt processing is provided in sections that follow.

13.12.5 IRET

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

13.12.6 PUSH and POP

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

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

```
PUSH R1       !indirect address: push working
              register 1 onto the stack!
POP 05H       !direct address: pop the top
              stack byte into register 5!
PUSH @R4      !indirect address: pop the top
              stack byte into the byte
              pointed to by working register 4!
PUSH @11H     !indirect address: push onto
              the stack the byte pointed to
              by register 17!
```

13.13 Interrupts

The Z8® recognizes six different interrupts from internal and external sources, including internal timer/counters, serial I/O, and Port 3 lines. Interrupts may be individually or globally enabled/disabled using the Interrupt Mask Register IMR (FBH) and may be prioritized for simultaneous interrupt resolution using the Interrupt Priority Register IPR (F9H). When enabled, interrupt request processing automatically vectors to the designated service routine. When disabled, an interrupt request may be polled to determine when processing is needed.

13.13.1 Interrupt Initialization

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

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

```
EI       !enable interrupts!
```

or by a register manipulation instruction such as

```
OR       IMR,#80H
```

To globally disable interrupts, execute the instruction

```
DI       !disable interrupts!
```

This will cause bit 7 of IMR to be reset.

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

13.13.2 Vectored Interrupt Processing

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

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

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

```
CONSTANT
              INT_MASK_3  =           00110000B
GLOBAL
IRQ3_service       PROCEDURE          ENTRY

                                      !service routine for IRQ3!

              PUSH IMR
                                      !interrupts were globally disabled during the interrupt machine cycle - no
                                      DI is needed prior to modification of IMR!

              AND    IMR,#INT_MASK_3  !disable all but IRQ4 & 5!

              EI
              !...!                   !service interrupt!
                                      !interrupts are globally enabled now — must disable them prior to
                                      modification of IMR!

              DI

              POP  IMR                !restore entry IMR!

              IRET

END IRQ3_service
```

Note: IRQ4 and IRQ5 are enabled by the above sequence after IRQ3_service only if their respective IMR bits = 1 on entry to IRQ3_service.

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

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

```
POP    FLAGS

              !FLAGS=@SP!

RET           !PC=@SP!
```

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

13.13.3 Polled Interrupt Processing

Disabled interrupt requests may be processed in a polled mode, in which the corresponding bits of the Interrupt Request Register (IRQ) are examined by the software. When an interrupt request bit is found to be a logic 1, the interrupt should be processed by the appropriate service routine. During such processing, the interrupt request bit in the IRQ must be cleared by the software in order for subsequent interrupts on that line to be distinguished from the current one. If more than one interrupt request is to be processed in a polled mode, polling should occur in the order of established priorities. For example, assume that IRQ0, IRQ1, and IRQ4 are to be polled and that established priorities are, from high to low, IRQ4, IRQ0, IRQ1. An instruction sequence like the following should be used to poll and service the interrupts:

```
!...!

!poll interrupt inputs here!

              TCM        IRQ, #00010000B         !IRQ4 need service?!
              JR         NZ,TESTO                !no!
              CALL       IRQ4_service            !yes!

TESTO TCM         IRQ, #00000001B         !IRQ0 need service?!
              JR         NZ,TEST1                        !no!
              CALL       IRQ0_service

TEST1         TCM        IRQ, #00000O10B         !IRQ1 need service ?!
              JR         NZ, DONE                        !no!
              CALL       IRQ1_service
DONE !...!

IRQ4_service      PROCEDURE       ENTRY

              !...!
              AND        IRQ, #11101111B         !clear IRQ4!
              !...!
              RET

END IRQ4_service

IRQ0_service      PROCEDURE       ENTRY

              !...!
              AND        IRQ, #11111110B         !clear IRQ0!
              !...!
              RET

END IRQ0_service

IRQ1_service      PROCEDURE       ENTRY
              !...!
              AND        IRQ, #11111101B         !clear IRQ1!
              !...!
              RET

END IRQ1_service
!...!
```

13.14 Timer/Counter Functions

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

- Internal Delay Timer
- Maintenance of a Time-Of-Day Clock
- Watch-Dog Timer
- External Event Counting
- Variable Pulse Train Output
- Duration Measurement of External Event
- Automatic Delay Following External Event Detection

Each timer/counter is driven by its own 6-bit prescaler, which is in turn driven by the internal Z8 clock divided by four. For T_1, the internal clock may be gated or triggered by an external event or may be replaced by an external clock input. Each timer/counter may operate in either single-pass or continuous mode where, at end-of-count, either counting stops or the counter reloads and continues counting. The counter and prescaler registers may be altered individually while the timer/counter is running; the software controls whether the new values are loaded immediately or when end-of-count (EOC) is reached.

Although the timer/counter prescaler registers (PRE0 and PRE1) are write-only, there is a technique by which the timer/counters may simulate a readable prescaler. This capability is a requirement for high resolution measurement of an event's duration. The basic approach requires that one timer/counter be initialized with the desired counter and prescaler values. The second timer/counter is initialized with a counter equal to the prescaler of the first timer/counter and a prescaler of 1. The second timer/counter must be programmed for continuous mode. With both timer/counters driven by the internal clock and started and stopped simultaneously, they will run synchronous to one another; thus, the value read from the second counter will always be equivalent to the prescaler of the first.

13.14.1 Time/Count Interval Calculation

To determine the time interval (i) until EOC, the equation

$$i = t \times p \times v$$

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

$$1/(XTAL/8)$$

(assumes internal clock set for XTAL divide by 2 mode)

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

$$1\text{us} \times 1 \times 1 \leq i \leq 1\text{us} \times 64 \times 256$$
$$1\text{us} \leq i \leq 16.384 \text{ ms}$$

To determine the count (c) until EOC for T_1 with external clock input, the equation

$$c = p \times v$$

characterizes the relation between the T_1 prescaler (p) and the T_1 counter (v). The divide-by-8 on the input frequency is bypassed in this mode. The count range is

$$1 \times 1 \leq c \leq 64 \times 256$$
$$1 \leq c \leq 16{,}384$$

13.14.2 T_{OUT} Modes

Port 3, bit 6 (P36) may be configured as an output (T_{OUT}) which is dynamically controlled by one of the following

- T_0
- T_1
- Internal Clock

When driven by T_0 or T_1, T_{OUT} is reset to a logic 1 when the corresponding load bit is set in timer control register TMR (F1H) and toggles on EOC from the corresponding counter.

When T_{OUT} is driven by the internal clock, that clock is directly output on P36.

While programmed as T_{OUT}, P36 is disabled from being modified by a write to port register 03H; however, its current output may be examined by the Z8 software by a read to port register 03H.

13.14.3 T_{IN} Modes

Port 3, bit 1 (P31) may be configured as an input (T_{IN}) which is used in conjunction with T1 in one of four modes.

- External Clock Input
- Gate Input for Internal Clock
- Nonretriggerable Input for Internal Clock
- Retriggerable Input for Internal Clock

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

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

The 'external clock input' T_{IN} mode supports the counting of external events, where an event is seen as a High-to-Low transition on T_{IN}. Interrupt request IRQ5 is generated on the nth occurrence (single-pass mode) or on every nth occurrence (continuous mode) of that event.

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

The "nonretriggerable input" T_{IN} mode provides for auto matic delay timing following an external event. In this mode, T1 is loaded and clocked by the Z8 internal clock following the first High-to-Low transition on T_{IN} after T1 is enabled. T_{IN} transitions that occur after this point do not affect T_1. In single-pass mode, the enable bit is reset on EOC; further T_{IN} transitions will not cause T_1 to load and begin counting until the software sets the enable bit again. In continuous mode, EOC does not modify the enable bit, but the counter is reloaded and counting continues immediately; IRQ5 is generated every EOC until software resets the enable bit. This T_{IN} mode may be used, for example, to time the line feed delay following end of line detection on a printer or to delay data sampling for some length of time following a sample strobe.

The "retriggerable input" T_{IN} mode will load and clock T_1 with the Z8 internal clock on every occurrence of a High-to-Low transition on T_{IN}. T_1 will time-out and generate interrupt request IRQ5 when the programmed time interval (determined by T1 prescaler and load register values) has elapsed since the last High-to-Low transition on T_{IN}. In single-pass mode, the enable bit is reset on EOC; further T_{IN} transitions will not cause T1 to load and begin counting until the software sets the enable bit again. In continuous mode, EOC does not modify the enable bit, but the counter is reloaded and counting continues immediately; IRQ5 is generated at every EOC until the software resets the enable bit. This T_{IN} mode may provide such functions as watch-dog timer (in other words, interrupt if conveyor belt stopped or clock pulse missed), or keyboard time-out (in other words., interrupt if no input in x ms).

13.14.4 Examples

Several possible uses of the timer/counters are given in the following four examples.

13.14.5 Time-Of-Day Clock

The following module illustrates the use of T1 for maintenance of a time-of-day clock, which is kept in binary format in terms of hours, minutes, seconds, and hundredths of a second. It is desired that the clock be updated once every hundredth of a second; therefore,T1$_1$ is programmed in continuous mode to interrupt 100 times a second. Although T1 is used for this example, T0 is equally suited for the task.

The procedure for initializing the timer (TOD_INIT), the interrupt service routine (TOD) which updates the clock, and the interrupt vector forT1 end-of-count (IRQ_5) are illustrated below (XTAL = 7.3728 MHz, XTAL/2 mode is assumed):

```
Z8ASM 2.0

LOC    OBJ CODE      STMT  SOURCE  STATEMENT
                       1   TIMER1       MODULE
                       2   CONSTANT
                       3    HOUR=       R12
                       4    MINUTE      =       R13
                       5    SECOND      =       R14
                       6    HUND=       R15
                       7                $SECTION  PROGRAM
                       8   GLOBAL
                       9   !IRQ5 interrupt vector!
                      10                $ABS  10
P 0000 OOOF'          11   IRQ_5  ARRAY [1 WORD]      =        [TOD]
                      12
                      13                $REL
P 000C                14   TOD_INIT             PROCEDURE
                      15   ENTRY
P 0000 E6 F3 93       16                LD      PRE1,#10010011B
                      17                                         !bit 2-7 prescaler = 36;
                      18                                         bit 1  internal clock;
                      19                                         bit 0  continuous mode!
P 0003 E6 F2 00       20          LD    T1,#00H                  !(256) time-out =
                      21                                         1/100 second!
P 0006 46 F1 0C       22          OR    TMR,#OCH                 !load, enable T1!
P 0009 8F             23          DI
P 000A 46 FB 20       24          OR    IMR,#20H                 !enable T1 interrupt!
P 000D 9F             25          EI
P 000E AF             26          RET
P 000F                27   END    TOD_INIT
                      28
P 000F                29   TOD    PROCEDURE
                      30   ENTRY
P 000F 70 FD          31          PUSH  RP
                      32                                         !Working register file 10H to 1FH contains
                      33                                         the time of day clock!
P 0011 31 10          34          SRP   #10H
P 0013 FE             35          INC   HUND                     !1 more .01 sec!
P 0014 A6 EF 64       36          CP    HUND,#64H                !full second yet?!
P 0017 EB 13          37          JR    NE,TOD_EXIT              !jump if no!
P 0019 BO EF          38          CLR   HUND
P 001B EE             39          INC   SECOND                   !1 more second!
P 001C A6 EE 3C       40          CP    SECOND,#3CH              !full minute yet?!
P 001F EB OB          41          JR    NE,TOD_EXIT              !jump if no!
P 0021 BO EE          42          CLR   SECOND
```

```
P 0023 DE          43           INC    MINUTE                  !1 more minute!
P 0024 A6 ED 3C    44           CP     MINUTE,#3CH             !full hour yet?!
P 0027 EB O3       45           JR     NE,TOD_EXIT             !jump if no!
P 0029 BO ED       46           CLR    MINUTE
P 002B CE          47           INC    HOUR
                   48     TOD_EXIT:

P 002C 50 FD       49           POP    RP                      !restore entry RPl
P 002E BF          50           IRET
P 002F             51     END   TOD
                   52     END   TIMER1

 0 ERRORS
ASSEMBLY  COMPLETE

TOD_INIT                  TOD
 7 instructions           17 instructions
 15 bytes                 32 bytes
 16 us                    19.5 us (average) including interrupt response time
```

13.14.6 Variable Frequency, Variable Pulse Width Output

The following module illustrates one possible use of T_{OUT}. Assume it is necessary to generate a pulse train with a 10 percent duty cycle, where the output is repetitively high for 1.6 ms and then low for 14.4 ms. To do this, T_{OUT} is controlled by end-of-count from T1, although T0 could alternately be chosen. This examples makes use of the Z8 feature that allows a timer's counter register to be modified without disturbing the count in progress. In continuous mode, the new value is loaded when T1 reaches EOC. T1 is first loaded and enabled with values to generate the short interval. The counter register is then immediately modified with the value to generate the long interval; this value is loaded into the counter automatically on T1 EOC. The prescaler selected value must be the same for both long and short intervals. Note that the initial loading of the T1 counter register is followed by setting the T1 load bit of timer control register TMR (F1H); this action causes T_{OUT} to be reset to a logic 1 output. Each subsequent modification of the T1 counter register does not affect the current T_{OUT} level, since the T1 load bit is NOT altered by the software. The new value is loaded on EOC and T_{OUT} will toggle at that time. The T1 interrupt service routine should simply modify the T1 counter register with the new value, alternating between the long and short interval values.

In the example which follows, bit 0 of register 04H is used as a software flag to indicate which value was loaded last. This module illustrates the procedure for T1/T_{OUT} initialization (PULSE_INIT), the T1 interrupt service routine (PULSE), and the interrupt vector forT1$_1$ EOC (IRQ_5). XTAL = 8 MHz, XTAL/2 mode is assumed.

```
Z8ASM 2.0

LOC    OBJ CODE      STMT  SOURCE STATEMENT
                       1   TIMER2       MODULE
                       2         $SECTION PROGRAM
                       3   GLOBAL
                       4                                !IRQ5 interrupt vector!
                       5         ABS    10
P 0000 0017'           6   IRQ_5 ARRAY[1 WORD]     =    [PULSE]
                       7
                       8         $REL
P 000C                 9   PULSE_INIT   PROCEDURE
                      10   ENTRY
P 000 E6 F3 03        11         LD     PRE1,#00000011B
                      12                                !bit 2-7 prescaler= 64;
                      13                                bit 1 internal clock;
                      14                                bit 0 continuous mode!
P 0003 E6 F7 00       15         LD     P3M,#00H        !bit 5: P36 = output (TOUT)!
P 0006 E6 F2 19       16         LD     T1,#19H         !for short interval!
P 0009 8F             17         DI
P 000A 46 FB 20       18         OR     IMR,#00100000B  !enable T1 interrupt!
P 000D E6 F1 8C       19         LD     TMR,#10001100B
                      20                                !bit 6-7 Tout controlled
                      21                                by T1;
                      22                                bit 3 enable T1;
                      23                                bit 2 load T1!

                      24   !Set long interval counter, to be loaded on T1 EOC!
P 0010 E6 F2 E1       25         LD     T1,#0E1H

                      26   !Clear alternating flag for PULSE!
P 0013 B0 04          27         CLR    04H             != 0  25 next;
                      28                                 = 1  225 next!
P 0015 9F             29         EI
P 0016 AF             30         RET
```

```
P 0017              31     END    PULSE_INIT
                    32
                    33
P 0017              34     PULSE PROCEDURE
                    35     ENTRY
P 0017 E6 F2 E1     36            LD     T1,#0E1H          !new load value!
P 001A B6 04 01     37            XOR    04H,#01H          !which value next?!
P 001D 6B 03        38            JR     Z,PULSE_EXIT      !should be 225!
P 001F E6 F2 19     39            LD     T1,#19H           !should be 25!
                    40     PULSE_EXIT
P 0022 BF           41            IRET
P 0023              42     END    PULSE
                    43     END    TIMER2

0 ERRORS
ASSEMBLY  COMPLETE
```

PULSE_INIT	PULSE
10 instructions	5 instructions
23 bytes	12 bytes
23 us	25 us (average) including interrupt response time

13.14.7 Cascaded Timer/Counters

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

To determine the time interval (i) until EOC, the equation

$$i = t \times p0 \times v0 \times (2 \times p1 \times v1 - 1)$$

characterizes the relation between the T0 prescaler (p0) and counter (v0), the T1 prescaler (p1) and counter (v1), and the clock input period (t). Assuming XTAL = 8 MHz, the measurable time interval range is:

$$1 \text{ us} \times 1 \times 1 \times (2 \times 1 - 1) \le i \le$$
$$1 \text{ us} \times 64 \times 256 \times (2 \times 64 \times 256 - 1)$$
$$1 \text{ us} \le i \le 536.854528 \text{ s}$$

Figure 13-3 illustrates the interconnection between T0 and T1. The following module illustrates the procedure required to initialize the timers for a 1.998 second delay interval

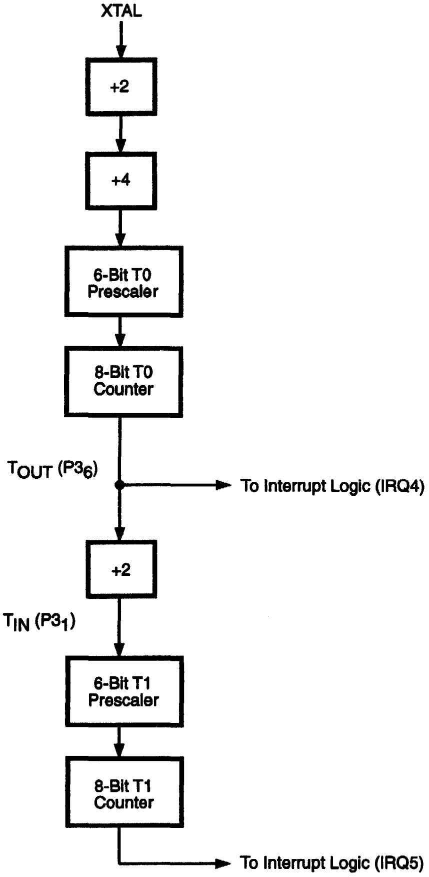


Figure 13-3. Cascaded Timer/Counters

```
Z8ASM 2.0

LOC     OBJ CODE      STMT  SOURCE  STATEMENT
                        1   TIMER3        MODULE
                        2   GLOBAL
P 0000                  3   TIMER_16              PROCEDURE
                        4   ENTRY
P 0000 E6 F3 28         5          LD             PRE1,#00101000B
                        6                                           !bit 2-7 prescaler = 10;
                        7                                           bit 1 external clock;
                        8                                           bit 0 single-pass mode!
P 0003 E6 F7 00         9          LD             P3M,#00H          !bit 5 let P36 be Tout!
P 0006 E6 F2 64        10          LD             T1,#64H           !T1 counter register!
P 0009 E6 F5 29        11          LD             PRE0,#00101001B   12
                                                                    !bit 2-7 prescaler = 10;
                       13                                            bit 0 continuous mode!
P 000C E6 F4 64        14          LD             T0,#64H           !T0 counter register!
P 000F 8F              15          DI
P 0010 56 FB 2B        16          AND            IMR,#00101011B    !disable IRQ2 (Tin);
                       17                                           and IRQ4 (T0)!
P 0013 46 FB 20        18          OR             IMR,#00100000B    !enable IRQ5 (T1)!
P 0016 9F              19          EI
P 0017 E6 F1 4F        20          LD             TMR,#01001111B
                       21                                           !bit6-7TOUTcontrolled
                       22                                           by T0;
                       23                                           bit4-5TINmodeisext.
                       24                                           clockinput;
                       25                                           bit 3 enable T1;
                       26                                           bit 2 load T1;
                       27                                           bit 1 enable T0;
                       28                                           bit 0 enable T0!
P 001A AF              29          RET
P 001B                 30   END    TIMER_16
                       31   END    TIMER3

0 ERRORS
ASSEMBLY  COMPLETE

11 instructions
27 bytes
26.6 us
```

13.14.8 Clock Monitor

T1 and T_{IN} may be used to monitor a clock line (in a diskette drive, for example) and generate an interrupt request when a clock pulse is missed. To accomplish this, the clock line to be monitored is wired to P3$_1$ (T_{IN}). T_{IN} should be programmed as a retriggerable input to T_1, such that each falling edge on T_{IN} will cause T1 to reload and continue counting. If T1 is programmed to time-out after an interval of one-and-a-half times the clock period being monitored, T1 will time-out and generate interrupt request IRQ5 only if a clock pulse is missed.

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

```
Z8ASM 2.0

LOC    OBJ CODE       STMT  SOURCE  STATEMENT
                       1    TIMER4        MODULE
                       2            $SECTION PROGRAM
                       3    GLOBAL
                       4                                         !IRQ5 interrupt vector!
                       5            $ABS   10
P 0000 0015'           6    IRQ_5   ARRAY[1 WORD]      =        [CLK_ERR]
                       7
                       8            $REL
P 000C                 9    MONITOR_INITPROCEDURE
                      10    ENTRY
P 0000 E6 F3 04       11            LD      PRE1,#00000100B
                      12                                        !bit 2-7 prescaler =1;
                      13                                        bit 1 external clock;
                      14                                        bit 0 single-pass mode!
P0003E6F700   15                    LD      P3M,#00H            !bit5letP36beT_OUT!
P0006E6F203   16                    LD      T1,#03H             !T1loadregister,
                      17                                        =1.5*2usec!
P 0009 8F             18            DI
P000A56FB3B   19                    AND     IMR,#00111011B      !disableIRQ2(T_IN)!
P000D46FB20   20                    OR      IMR,#00100000B      !enableIRQ5(T1)!
P00109F               21            EI
                      22
P 0011 E6 F1 38       23            LD      TMR,#00111000B
                      24                                        !bit4-5T_INmodeis
                      25                                        retrig. input;
                      26                                        bit 3 enable T1!
P 0014 AF             27            RET
P 0015                28    END     MONITOR_INIT
                      29
                      30
P 0015                31    CLK_ERR  PROCEDURE
                      32    ENTRY
                      33            !...!                       !handle the missed clock!
                      34
                      35    !if clock monitoring should continue...!
```

```
P 0015 46 F1 08     36          OR     TMR, #00001000B
                    37                                        !bit 3: enable T1 !
P 0018 BF           38          IRET
P 0019              39    END   CLK_ERR
                    40    END   TIMER4
0 ERRORS
ASSEMBLY  COMPLETE
MONITOR_INIT        CLK_ERR
 9 instructions     2+ instructions
 21 bytes           4+ bytes
 21.5 us            18.5 us+ including interrupt response time
```

13.15 I/O FUNCTIONS

The Z8® provides up to 32 I/O lines mapped into registers 0-3 of the internal register file. Each nibble of Port 0 is individually programmable as input, output, or address/data lines (A15-A12, A11-A8). Port 1 is programmable as a byte entity to provide input, output, or address/data lines (AD7-AD0). The operating modes for the bits of Ports 0 and 1 are selected by control register P01M (F8H). Selection of I/O lines as address/data lines supports access to external program and Data Memory. Each bit of Port 2 is individually programmable as an input or an output bit. Port 2 bits programmed as outputs may also be programmed (via bit 0 of P3M) to all have active pull-ups or all be open-drain (active pull-ups inhibited). In Port 3, four bits (P30-P33) are fixed as inputs, and four bits (P34-P37) are fixed as outputs, but their functions are programmable. Special functions provided by Port 3 bits are listed in Table 13-4.

Note: I/O feature options are device dependent. Consult the selected Z8 device product specification for exact I/O features available.

Table 13-4. Generic Z8 MCU Port 3 Special Functions

FUNCTION	BIT	SIGNAL
Handshake	P31	$\overline{DAV2}$/RDY2
	P32	$\overline{DAV0}$/RDY0
	P32	DAV1/$\overline{RDY1}$
	P34	RDY1/$\overline{DAV1}$
	P35	RDY0/$\overline{DAV0}$
	P36	RDY2/$\overline{DAV2}$
Interrupt Request	P30	IRQ3
	P31	IRQ2
	P32	IRQ0
	P33	IRQ1
Counter/ Timer	P31	T_{IN}
	P36	T_{OUT}
Data Memory Select Status Out	P34	$\overline{DM}$
Serial I/O	P30	Serial In
	P37	Serial Out

13.15.1 Asynchronous Receiver/Transmitter Operation

In some cases, full-duplex, serial asynchronous receiver/transmitter operation is provided using P37 (output) and P30 (input) in conjunction with control register SIO (F0H), SIO is actually two registers: a receiver buffer and a transmitter buffer. Counter/Timer T0 provides the clock for control of the bit rate.

The Z8® always receives and transmits eight bits between start and stop bits. However, if parity is enabled, the eighth bit (D7) is replaced by the odd-parity bit when transmitted and a parity-error flag (= 1 if error) when received. Table 13-5 illustrates the state of the parity bit/parity error flag during serial I/O with parity enabled.

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

Table 13-5. Serial I/O With Odd Parity

Character Loaded Into SIO	Transmitted To Serial Line	Received From Serial Line	Transferred Character To SIO	Note*
11000011	01000011	01000011	01000011	no error
11000011	01000011	01000111	11000111	error
01111000	11111000	11111000	01111000	no error
01111000	11111000	01111000	11111000	error

* Left most bit is D7

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

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

To enable P30 and P37 for serial I/0, bit 6 of P3M (F7H) is set. To enable odd parity, bit 7 of P3M is set; to disable it, the bit is reset. For example, the instruction:

```
LD      P3M,#40H
```

will enable serial I/0, but disable parity. The instruction:

```
LD      P3M,#0C0H
```

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

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

$$\text{bit rate} = \frac{\text{input clock frequency}}{(2\text{x}4\text{x}T0 \text{ prescaler x } T0 \text{ counter x } 16)}$$

The final divide-by-16 is incurred for serial communications, since in this mode T0 runs at 16 times the bit rate in order to synchronize the data stream. To configure the Z8 for a specific bit rate, appropriate values must first be selected for T0 prescaler and T0 counter by the above equation; these values are then programed into registers T0 (F4H) and PRE0 (F5H) respectively. Note that PRE0 also controls the continuous vs. single-pass mode for T0; continuous mode should be selected for serial I/O. For example, given an input clock frequency of 7.3728 MHz and a selected bit rate of 9600 bits per second, the equation is satisfied by T0 counter = 2 and prescaler = 3. The following code sequence will configure the T0 counter and T0 prescaler registers:

```
LD      T0,#02H          !T0 counter = 2!
LD      PRE0,#00001101B
                         !bit 2-7 prescaler = 3; bit 0
                         continuous mode!
```

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

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

```
        OR      TMR,#03H
```

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

Characters to be output to the serial line should be written to serial I/O register SIO (F0H). IRQ4 will be generated when all bits have been transferred out.

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

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

```
Z8ASM 2.0

LOC    OBJ CODE      STMT  SOURCE  STATEMENT
                       1   SERIAL_IO    MODULE
                       2   CONSTANT
                       3    next_addr    =       41H
                       4    start   =       42H
                       5    length =        1EH
                       6   $SECTION  PROGRAM
                       7   GLOBAL
                       8                                      !IRQ3 vector!
                       9                $ABS   6
P 0006 000'           10   IRQ_3  ARRAY [1 WORD] =    [GET_CHARACTER]
                      11
                      12                $REL   0
P 0000                13   GET_CHARACTER      PROCEDURE  ENTRY
                      14
                      15                                      !Serial I/O receive interrupt service!
                      16                                      !Echo received character and wait for
                      17                                      echo completion!
P 0000 E4 F0 F0       18                ld       SIO,SIO      !echo!
                      19
                      20                                       !Save it in circular buffer!
P 0003 F5 F0 41       21                ld       @next_addr,SIO !save in buffer!
P 0006 20 41          22                inc      next_addr    !Point to next position!
P 0008 A6 41 60       23                cp       next_addr,#start+length
                      24                                      !Wrap-around yet?!
P 000B EB 03          25                jr       ne,echo_wait  !No.!
P 000D E6 41 42       26                ld       next_addr,#start!Yes.  Point to start!
                      27                                      !Now, wait for echo complete!
                      28   echo_wait
P 0010 66 FA 10       29                tcm      IRQ,#10H     !Transmitted yet?!
P 0013 EB FB          30                jr       nz,echo_wait !Not yet!
                      31
P 0015 56 FA EF       32                and      IRQ,#0EFH    !Clear IRQ4!
```

```
P 0018 BF                 IRET                          !Return from interrupt!
P 0019              END   GET_CHARACTER
                    END   SERIAL_IO

0 ERRORS
ASSEMBLY COMPLETE

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

13.15.2 Automatic Bit-Rate Detection

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

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

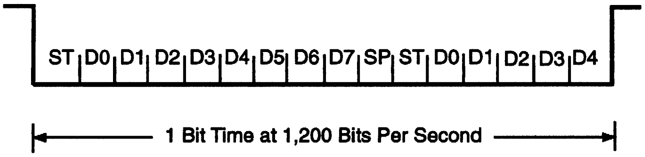
$$n = 19{,}200/b$$

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

The following example is capable of distinguishing between the bit rates shown in Table 13-6 and assumes an input clock frequency of 7.3728 MHz, a T0 prescaler of 3, XTAL/2 mode, and serial I/O enabled with parity disabled. This example requires that a character with its low order bit = 1 (such as a carriage return) be sent to the serial channel. The start bit of this character can be measured by counting the number of zero bits collected before the low order 1 bit. The number of zero bits actually collected into data bits by the serial channel is less than n (as given in the above equation), due to the detection of start and stop bits. Figure 13-4 illustrates the collection (at 19,200 bits per second) of a zero bit transmitted to the Z8 at 1,200 bits per second. Notice that only 13 of the 16 zero bits received are collected as data bits.

Table 13-6. Inputs to the Automatic Bit Rate Detection Algorithm

Bit Rate	Number of Bits Received Per Bit Transmitted	Number of Bits Collected as Data Bits		T$_0$ Counter	
		dec	binary	dec	binary
19200	1	0	00000000	1	00000001
9600	2	1	00000001	2	00000010
4800	4	3	00000011	4	00000100
2400	8	7	00000111	8	00001000
1200	16	13	00001101	16	00010000
600	32	25	00011001	32	00100000
300	64	49	00110001	64	01000000
150	128	97	01100001	128	10000000



ST = Start Bit Sp = Stop Bit Dn = Data Bit n

Each Interval Shown = 1 Bit Time
At 19,200 Bits Per Second

Figure 13-4. Collection of a Start Bit Transmitted at 19.2 KBps

Once the number of zero bits in the start bit has been collected and counted, it remains to translate this count into the appropriate T0 counter value and program that value into T0 (F4H). The patterns shown in the two binary columns of Table 13-6 are utilized in the algorithm for this translation.

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

(9 x n)/b

where n and b are as defined above. The following module illustrates a sample program for automatic bit rate detection.

```
Z8ASM 2.0

LOC     OBJ CODE      STMT  SOURCE  STATEMENT
                        1   BIT_RATE      MODULE
                        2   EXTERNAL
                             3      DELAY PROCEDURE
                        4   GLOBAL
P 0000                  5   main    PROCEDURE
                        6           ENTRY
P 0000 8F               7           di                        !Disable interrupts!
P 0001 56 FB 77         8           and     IMR,#77H          !IRQ3 polled mode!
P 0004 56 FA F7         9           and     IRQ,#0F7H         !Clear IRQ3!
P 0007 E6 F7 40        10           ld      P3M,#40H          !Enable serial I/O!
P 000A E6 F4 01        11           ld      T0,#01H
p 000D E6 F5 OD        12           ld      PRE0,#(3 SHL 2)+1 !bit rate = 19,200;
                       13                                     continuous count mode!
P 0010 B0 E0           14           clr     R0                !Init. zero byte counter!
P 0012 E6 F1 03        15           ld      TMR#03H           !Load and enable TO!
                       16
                       17   !Collect input bytes by counting the number of null
                       18   characters received.  Stop when non-zero byte received!
                       19   collect:
P 0015 76 FA 0 8       20           tm      IRQ,#08H          !Character received?!
P 0018 6B FB           21           jr      Z,collect         !Not yet!
P 001A 18 F0           22           ld      R1,SIO !Get the character!
P 001C 56 FA F7        23           and     IRQ,#0F7H         !Clear interrupt request!
P 001F 1E              24           inc     R1                !Compare to 0 ...!
P 0020 1A 05           25           djnz    R1,bitloop        !...(IN 3 bytes of code)!
P 0022 06 E0 08        26           add     R0,#08H           !Update count of 0 bits!
P 0025 8B EE           27           jr      collect
                       28   bitloop:                          !Add in zero bits from low
                       29                                     end of 1st non-zero byte!
P 0027 E0 E1           30           RR      R1
P 0029 7B 03           31           jr      c,count_done
P 002B OE              32           inc     R0
P 002C 8B F9           33           jr      bitloop
                       34
                       35                                     !R0 has number of zero bits collected!
                       36                                     !Translate R0 to the appropriate T0 counter value!
                       37   count_done                        !R0 has count of zero bits!
```

```
P 002E 1C 07        38              ld      R1,#07H
P 0030 2C 80        39              ld      R2,#80H         !R2 will have T0 counter value!
P 0032 90 E0        40              RL      R0
                    41
P 0034 90 E0        42 loop:        RL      RO
P 0036 7B 04        43              jr      c,done
P 0038 E0 E2        44              RR      R2
P 003A 1A F8        45              djnz    r1,loop
                    46
P 003C 29 F4        47      done    ld      T0,R2           !Load value for detected
                    48                                      bit rate!
                    49                                      !Delay long enough to clear serial line of bit stream!
P 003E D6 0000*     50              call    DELAY
                    51                                      !Clear receive interrupt request!
P 0041 56 FA F7     52              and     IRQ,#0F7H
                    53
P 0044              54      END     main
                    55      END     bit_rate

0 ERRORS
ASSEMBLY  COMPLETE

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

13.15.3 Port Handshake

Each of Ports 0, 1 and 2 may be programmed to function under input or output handshake control. Table 13-7 defines the port bits used for handshaking and the mode bit settings required to select handshaking. To input data under handshake control, the Z8® should read the input port when the $\overline{\text{DAV}}$ input goes Low (signifying that data is available from the attached device). To output data under handshake control, the Z8 should write the output port when the RDY input goes Low (signifying that the previously output data has been accepted by the attached device). Interrupt requests IRQ0, IRQ1, and IRQ2 are generated by the falling edge of the handshake signal input to the Z8 for Port 0, Port 1, and Port 2 respectively. Port handshake operations may therefore be processed under interrupt control.

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

```
        CLR     P2M     !Port 2 mode register all Port 2 bits
                        are outputs!

        OR      03H,#40H

                        !set DAV2 data not available!

        LD      P3M,#20H

                        !Port 3 mode register enable
                        Port 2 handshake!

        LD      02H,DATA

                        !output first data byte; DAV2 will
                        be cleared by the Z8 to indicate
                        data available to the peripheral
                        device!
```

Note that following the initialization of the output sequence, the software outputs the first data byte without regard to the state of the RDY2 input; the Z8 will automatically hold $\overline{\text{DAV2}}$ High until the RDY2 input is High. The peripheral device should force the Z8 RDY2 input line Low after it has latched the data in response to a Low on $\overline{\text{DAV2}}$. The Low on RDY2 will cause the Z8 to automatically force $\overline{\text{DAV2}}$ High until the next byte is output. Subsequent bytes should be output in response to interrupt request IRQ2 (caused by the High-to-Low transition on RDY2) in either a polled or an enabled interrupt mode.

Table 13-7. Port Handshake Selection

	Port 0	Port 1	Port 2

Input handshake lines	P32 = $\overline{\text{DAV}}$
	P35 = RDY
		P33 = $\overline{\text{DAV}}$
		P34 = RDY
			P31 = $\overline{\text{DAV}}$
			P36 = RDY

Output handshake lines	P32 = $\underline{\text{RDY}}$
	P35 = DAV
		P33 = $\underline{\text{RDY}}$
		P34 = DAV
			P31 = $\underline{\text{RDY}}$
			P36 = DAV

To select input handshake	set bit 6 & reset bit 7 of P01M (program high nibble as input)	set bit 3 & reset bit 4 of P01M (program byte as input)	set bit 7 of P2M (program high bit as input)
To select output handshake	set bits 6, 7 of P01M (program high nibble as output)	set bit 3, 4 of P01M (program byte as output)	set bit 7 of P2M (program high bit as output)
To enable handshake	set bit 5 of Port 3 (P3$_5$); set bit 2 of P3M	set bit 4 of Port 3 (P3$_4$); set bits 3, 4 of P3M	set bit 6 of Port 3 (P3$_6$); set bit 5 of P3M

13.16 ARITHMETIC ROUTINES

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

13.16.1 Binary to Hex ASCII

The following module illustrates the use of the ADD and SWAP arithmetic instructions in the conversion of a 16-bit binary number to its hexadecimal ASCII representation.

The 16-bit number is viewed as a string of four nibbles and is processed one nibble at a time from left to right, beginning with the high-order nibble of the lower memory address. 30H is added to each nibble if it is in the range 0 to 9; otherwise 37H is added. In this way, 00H is converted to 30H, 1H to 31H, ... 0AH to 41H, ... 0FH to 46H. Figure 13-5 illustrates the conversion of RR0 (contents = F2BEH) to its hex ASCII equivalent; the destination buffer is pointed to by RR4.

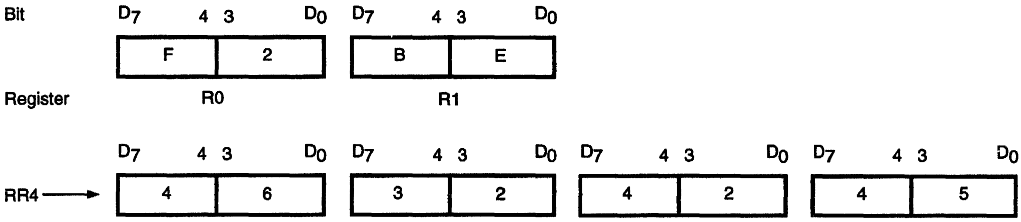


Figure 13-5. Conversion of (RR0) To Hex ASCII

```
Z8ASM 2.99     INTERNAL RELEASE
LOC    OBJ CODE      STMT  SOURCE  STATEMENT
                       1   ARITH  MODULE
                       2   GLOBAL
P 0000                 3   BINASC        PROCEDURE
                       4   !****************************************************
                       5   Purpose =     To convert a 16-bit binary
                       6                 number to Hex ASCII
                       7
                       8   Input=        RR0 =  16-bit binary number.
                       9                 RR4 =  pointer to destination
                      10                        buffer in external memory.
                      11
                      12   Output=       Resulting ASCII string (4 bytes)
                      13                 in destination buffer.
                      14                 RR4 incremented by 4 .
                      15                 RO, R2, R6 destroyed.
                      16   ****************************************************!
                      17   ENTRY
                      18
P 0000 6C 04          19          ld     R6,#04H            !nibble count!
P 0002 F0 E0          20   again: SWAP   R0                 !look at next nibble!
P 0004 28 E0          21          ld     R2,R0
P 0006 56 E2 OF       22          and    R2,#0FH            !isolate 4 bits!
                      23                                    !convert to ASCII  R2 + #30H if R0 in range 0
                                                            to 9
                      24                                    else R2 + #37H (in range 0A to 0F)!
```

```
P 0009 06 E2 30              ADD    R2,#30H
P 000C A6 E2 3A              cp     R2,#3AH
P 000F 7B 03                 jr     ult,skip
P 0011 06 E2 07              ADD    RS,#07H
P 0014 92 24         skip:   lde    @RR4,R2        !save ASCII in buffer!
P 0016 A0 E4                 incw   RR4            !point to next
                                                   buffer position!
P 0018 A6 E6 03              cp     R6,#03H        !time for second byte?!

P 001B EB 02                 jr     ne,same_byte   !no.!
P 001D 08 E1                 ld     RO,R1          !2nd byte!
                     same_byte:
P 001F 6A E1                 djnz   R6,again
P 0021 AF                    ret
P 0022               END     BINASC
                     END     ARITH

0 ERRORS
ASSEMBLY  COMPLETE
16 instructions
34 bytes
120.5 us (average)
```

13.16.2 BCD Addition

The following module illustrates the use of the add with carry (ADC) and decimal adjust (DA) instructions for the addition of two unsigned BCD strings of equal length. Within a BCD string, each nibble represents a decimal digit (0-9). Two such digits are packed per byte with the most significant digit in bits 7-4. Bytes within a BCD string are arranged in memory with the most significant digits stored in the lowest memory location. Figure 13-6 illustrates the representation of 5970 in a 6-digit BCD string, starting in register 33H.

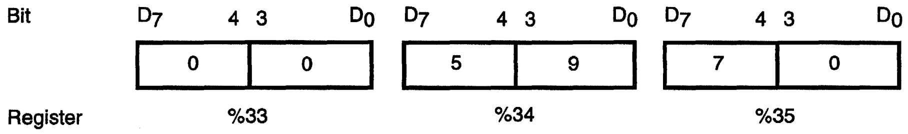


Figure 13-6. Unsigned BCD Representation

```
Z8ASM 2.0
LOC     OBJ CODE     STMT  SOURCE  STATEMENT
                       1   ARITH  MODULE
                       2   CONSTANT
                       3    BCD_SRC =   R1
                       4    BCD_DST =   R0
                       5    BCD_LEN =   R2
                       6   GLOBAL
P 0000                 7   BCADDPROCEDURE
                       8   !****************************************************
                       9   Purpose =      To add two paced BCD strings of
                      10                  equal length.
                      11                  dst <— dst + src
                      12
                      13   Input=         R0 = pointer to dst BCD string.
                      14                  R1 = pointer to src BCD string.
                      15                  R2 = byte count in BCD string
                      16                     (digit count = (R2)*2 ).
                      17
                      18   Output =               BCD string pointed to by R0 is
                      19                          the sum.
                      20                  Carry FLAG = 1 if overflow.
                      21                  R0 , R1 as on entry.
                      22                  R2 = 0
                      23   ****************************************************!
                      24   ENTRY
                      25
P 0000 02 12          26           add    BCD_SRC,BCD_LEN           !start at least...!
P 0002 02 02          27           add    BCD_DST,BCD_LEN           !significant digits!
P 0004 CF             28           rcf                              !carry = 0!
                      29   add_again:
P 0005 00 E1          30           dec    BCD_SRC                   !point to next two
                      31                                             src digits!
P 0007 00 E0          32           dec    BCD_DST                   !point to next two
                      33                                             dst digits!
P 0009 E3 31          34           ld     R3,@BCD_SRC               !get src digits!
P 000B 13 30          35           ADC    R3,@BCD_DST               !add dst digits!
P 000D 40 E3          36           DA     R3                        !decimal adjust!
P 000F F3 03          37           ld     @BCD_DST,R3               !move to dst!
P 0011 2A F2          38           djnz   BCD_LEN,add_again         !loop for next
                      39                                             digits!
P 0013 AF             40           ret                              !all done!
                      41
P 0014                42   END     BCADD
                      43   END     ARITH

 0 ERRORS
ASSEMBLY  COMPLETE
11 instructions
20 bytes
Execution time is a function of the number of bytes (n) in input BCD string:  20 us + 12.5(n-1) us
```

13.16.3 Multiply

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

```
Z8ASM 2.99 INTERNAL RELEASE
LOC    OBJ CODE     STMT  SOURCE STATEMENT
                     1    ARITH  MODULE
                     2    CONSTANT
                     3     MULTIPLIER   = R1
                     4     PRODUCT_LO = R3
                     5     PRODUCT_HI  = R2
                     6     COUNT = R0
                     7    GLOBAL
P 0000               8    MULT  PROCEDURE
                     9!****************************************************
                    10    Purpose =       To perform an 8-bit by 8-bit unsigned
                    11                    binary multiplication.
                    12
                    13    Input=          R1 = multiplier
                    14                    R3 = multiplicand
                    15
                    16    Output =        RR2 = product
                    17                    R0   destroyed
                    18    ****************************************************!
                    19    ENTRY
P 0000 0C 09        20          ld    COUNT,#09H             !8 BITS + 1!
P 0002 B0 E2        21          clr   PRODUCT_HI             !INIT HIGH RESULT BYTE!
P 0004 CF           22          RCF                          !CARRY =0!
P 0005 C0 E2        23    LOOP: RRC   PRODUCT_HI
P 0007 CO E3        24          RRC   PRODUCT_LO
P 0009 FB 02        25          jr    NC,NEXT
P 000B 02 21        26          ADD   PRODUCT_HI,MULTIPLIER
P 000D 0A F6        27    NEXT: djnz  COUNT,LOOP
P OOOF AF           28          ret
P 0010              29    END   MULT
                    30    END   ARITH

 0 ERRORS
ASSEMBLY  COMPLETE
9 instructions
16 bytes
92.5 us (average)
```

13.16.4 Divide

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

```
Z8ASM 2.0
LOC    OBJ CODE    STMT  SOURCE STATEMENT
                     1   ARITH  MODULE
                     2   CONSTANT
                     3     COUNT = R0
                     4     DIVISOR = R1
                     5     DIVIDEND_HI  = R2
                     6     DIVIDEND_LO = R3
                     7   GLOBAL
P 0000               8   DIVIDE PROCEDURE
                     9   !****************************************************
                    10   Purpose =      To perform a 16-bit by 8-bit unsigned
                    11                  binary division.
                    12
                    13   Input =        R1 = 8-bit divisor
                    14                  RR2 = 16-bit dividend
                    15
                    16   Output =       R3 = 8-bit quotient
                    17                  R2 = 8-bit remainder
                    18                  Carry flag = 1 if overflow
                    19                             = 0 if no overflow
                    20   ****************************************************!
                    21   ENTRY
P 0000 OC 08        22         ld       COUNT,#08H           !LOOP COUNTER!
                    23
                    24                                        !CHECK IF RESULT WILL FIT IN 8 BITS!
P 0002 A2 12        25         cp       DIVISOR,DIVIDEND_HI
P 0004 BB 02        26         jr       UGT,LOOP             !CARRY = 0 (FOR RLC)!
                    27                                        !WON'T FIT.  OVERFLOW!
P 0006 DF           28         SCF                            !CARRY = 1!
P 0007 AF           29         ret
                    30
                    31   LOOP  !RESULT WILL FIT.  GO AHEAD WITH DIVISION!
P 0008 10 E3        32         RLC      DIVIDEND_LO          !DIVIDEND * 2!
P 000A 10 E2        33         RLC      DIVIDEND_HI
P 000C 7B 04        34         jr       c,subt
P 000E A2 12        35         cp       DIVISOR,DIVIDEND_HI
P 0010 BB 03        36         jr       UGT,next             !CARRY = 0!
P 0012 22 21        37   subt: SUB      DIVIDEND_HI,DIVISOR
P 0014 DF           38         SCF                            !TO BE SHIFTED INTO RESULT!
P 0015 0A F1        39   next: djnz     COUNT,LOOP           !no flags affected!
                    40
                    41                                        !ALL    DONE!
```

```
P 0017 10 E3              RLC    DIVIDEND_LO
                                               !CARRY= 0 no overflow!
P 0019 AF                 ret
P 001A              END DIVIDE
                    END ARITH

0 ERRORS
ASSEMBLY  COMPLETE
15 instructions
26 bytes
124.5 us (average)
```

13.17 Conclusion

This section has focused on ways in which the Z8® microcomputer can easily yet effectively solve various application problems. In particular, the many sample routines illustrated here should aid the user in applying the Z8 to greater advantage. The major features of the Z8 have been described so that the user can continue to expand and explore the repertoire of uses for the Z8.

CHAPTER 14
THIRD-PARTY SUPPORT TOOLS

In addition to Zilog tool offerings, an extensive list of third party suppliers offer a variety of software (XASM, C Compilers, Simulators/Debuggers), hardware emulator, and OTP programmer (single and gang) products.

14.1 Third-Party Support—Emulators/ Programmers

Data I/O (OTP Programmer)	(800) 332-8246
EmulationTechnologies (OTP Socket Adapters)	(408) 982-0660
iSystems	(49)8131-25085
Logical Devices, Inc. (OTP Programmer)	(800) 331-7766
Needham Electronics (OTP Programmer)	(916) 924-8037
Orion Instruments	(408) 747-0440
Signum Systems	(805) 371-4608
Systems General (OTP Programmer)	(408) 263-6667

14.2 Third-Party Support—Assemblers/C Compilers

2500AD Software	(719) 395-8683
Avocer Systems	(800) 448-8500
ByteCraft	(519) 888-6911
Micro Computer Control	(609) 466-1751
Production Languages Corp.	(817) 599-8363
Pseudo Corp.	(503) 683-9173

Z8® Microcontroller
Technical Description

I

Zilog Z8® Software

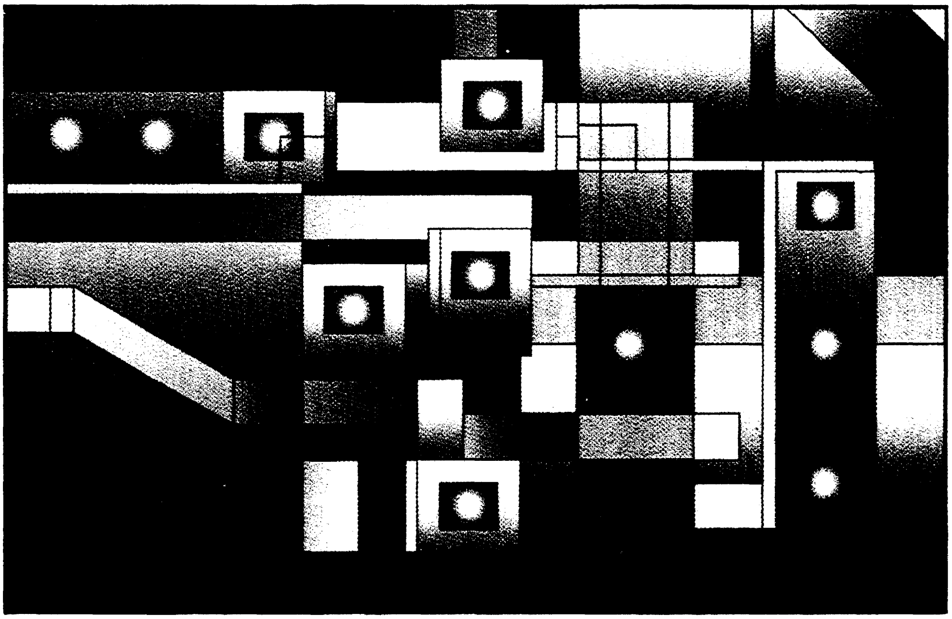
II

Zilog General
Information

III

ZiLOG

USER'S GUIDE



asm Z8®

CROSS ASSEMBLER

Related Publications

IEEE Proposal P695 Microprocessor Universal Format for Object Modules. IEEE Micro August 1983 Vol. 3 & 4 pp. 48-66

Super8 Technical Manual, document number 03-8257-0X

Universal Object Files Utilities User's Guide, document number 03-8236-0X

Z8 Microcomputer Technical Manual, document number 03-3047-0X

Trademark Acknowledgements

asmS8, cas, PLZ/ASM, Super8, System 8000, and Z8 are trademarks of Zilog Inc.
UNIX is a trademark of AT&T Bell Laboratories
VAX is a trademark of Digital Equipment Corporation

Zilog does not support the software mentioned in this publication, use at own risk.

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.

All specifications (parameters) are subject to change without notice. The applicable Zilog test documentation will specify which parameters are tested.

asm Z8® Cross Assembler User's Guide
Table of Contents

CHAPTER 1
OVERVIEW

1.1 INTRODUCTION

Zilog's Super8/Z8 Cross-Assembler (asmS8) takes a source file containing assembly language statements and translates it into a corresponding object file. It can produce a listing containing the source code, object code, and comments. The assembler supports macros and conditional assembly. It is written in C and runs on the UNIX operating system. Figure 1-1 illustrates the development path of a typical program.

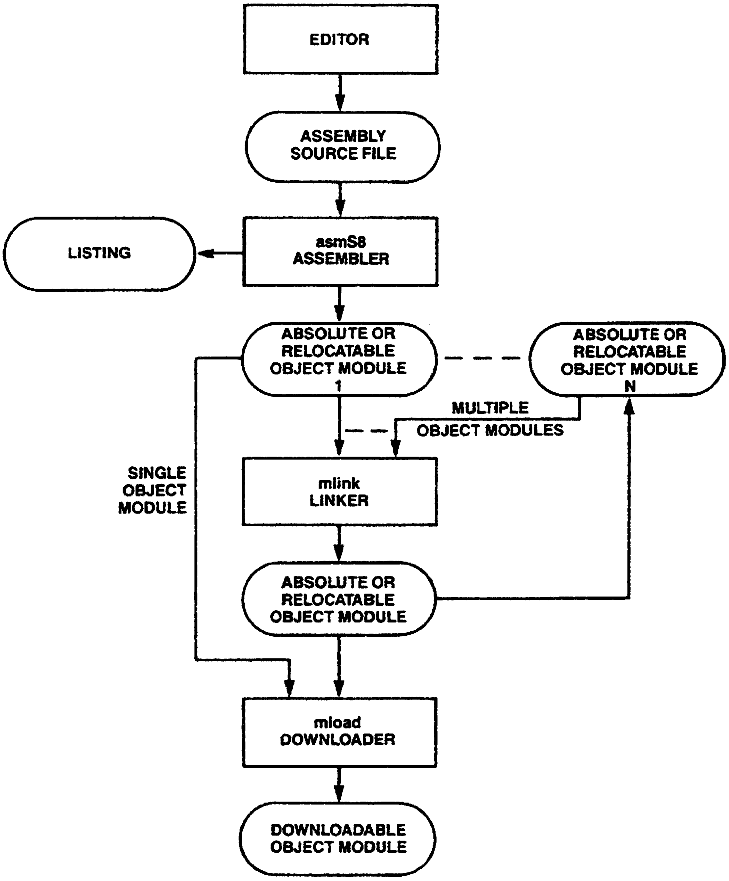

EDITOR
ASSEMBLY SOURCE FILE
LISTING
asmS8 ASSEMBLER
ABSOLUTE OR RELOCATABLE OBJECT MODULE 1
ABSOLUTE OR RELOCATABLE OBJECT MODULE N
MULTIPLE OBJECT MODULES
SINGLE OBJECT MODULE
mlink LINKER
ABSOLUTE OR RELOCATABLE OBJECT MODULE
mload DOWNLOADER
DOWNLOADABLE OBJECT MODULE

Figure 1-1. asmS8 Program Development Cycle

1.1 INTRODUCTION (Continued)

The assembler can produce relocatable and absolute object code. Object files can contain a mixture of absolute and relocatable code. Object files then can be linked with other object files and loaded into memory.

For a description of the architecture of the Super8 family of microcomputers, refer to the Super8 Technical Manual. For a description of the architecture of the Z8 family, refer to the Z8 Microcomputer Technical Manual.

1.2 ASSEMBLER OVERVIEW

The asmS8 Cross-Assembler is a macro assembler, written in C, that runs on the UNIX operating system for the DEC VAX and VMS, IBM-PC, and Zilog System 8000. The assembler produces output in a universal object code format (refer to the Universal Object Files Utilities User's Guide).

1.2.1 Assembler Enhancements

Providing more than compatibility with existing hardware and software, the asmS8 assembler includes new features not available in earlier assemblers. Integer arithmetic on numbers up to 80 bits long is supported, as is arbitrary integer arithmetic on external and relocatable symbols. Additional expression operators are defined, and syntax rules for expressions and operand delimiters have fewer restrictions.

The asmS8 assembler increases support for constants by providing floating-point constants in addition to those numbers supported in the C language. However, floating-point arithmetic in assembly-time expressions is not supported.

1.2.2 Modules

A program consists of one or more separately coded and assembled modules. Modules are referred to as either source modules or object modules.

A source module is made up of assembly language statements. These statements are then translated by the asmS8 assembler into an object module that can either be separately executed by the Super8 (or Z8) microprocessor, or linked with other object modules to form a complete program. The user can also control the operation of the assembler by including assembler directives, or "pseudo-ops," in the source code. Briefly, pseudo-ops resemble opcodes in form, but not function (see Chapter 3).

Depending on the assembler directives used, addresses within an object module or program can be absolute (addresses in the source program correspond exactly to logical memory addresses) or relocatable (addresses can be assigned relative to a logical base address at a later time). Object modules should be made relocatable whenever possible. This facilitates the ability to link with other object modules and also provides the ability to load object programs anywhere in memory. Relocatable addressing also allows the creation of libraries of commonly used procedures (including math or input/output routines) that can be linked selectively into several programs as desired.

1.3 RELOCATION AND LINKING

Relocation refers to the ability to bind a program module and its data to a particular memory area after the assembly process. The output of the assembler is an object module that contains enough information to allow the linker to assign that module to a memory area. Since many modules can be loaded together to form a complete program, a need for inter-module communication arises. For example, one module can contain a call to a routine that was assembled as part of another module and is located in some arbitrary part of memory. Therefore, the assembler must provide information in the object module that allows the linker to link inter-module references.

There are several major advantages to using the relocating assembler as compared to an absolute assembler:

1) Assignment of modules to memory areas can be handled by the linker rather than requiring the programmer to assign fixed absolute locations via the "ORG" pseudo-op; thus, modules can be relocated without requiring reassembly.

2) If errors are found in one module, only that one module needs to be reassembled and relinked with the other modules, thus increasing software productivity.

3) Programs can be structured into independent modules, coded separately and assembled, even though other modules may not yet exist.

4) Libraries of commonly used modules can be built and then linked with programs without requiring reassembly of the library module.

5) Communications between overlay segments can be achieved through methods similar to normal (non-overlay) inter-module references.

Unless otherwise specified, the output of the assembler is in relocatable form. During program execution, the instruction will be located at the memory location

1.3 RELOCATION AND LINKING (Continued)

specified by the linker (assigned origin plus the relative offset). Thus, a relocatable module has its first instruction located at the memory location that is the assigned origin of the module as determined by the linker.

To achieve relocation, addresses are altered at linkage time for both instructions that reference memory locations and data values that serve as pointers to memory locations. This process is transparent to the programmer.

1.3.1 Inter-Module References

The asmS8 assembler supports two pseudo-ops (or pseudo operation codes), GLOBAL and EXTERNAL, so that instructions can refer to "names" (either data values or entry points) in other assembled modules. GLOBAL means that the listed names are defined in this module and are available for use by other modules. EXTERNAL means that the names are used in this module, but are defined in another module where they are declared to be global. The syntax requires one or more names to follow either pseudo-op.

The GLOBAL name can be either absolute or relocatable. A portion of the object module contains a list of both the GLOBALs that are defined in the module, and the EXTERNALs that the module references. One function of the linker is to match all the EXTERNALs with the appropriate GLOBALs so that every instruction will reference the correct address during program execution.

A more thorough discussion of pseudo-ops is given in Chapter 3.

1.3.2 Sections

Programs can be divided into sections that are mapped into various areas of memory when linked or loaded for execution. A single module can contain several sections, each allocated to a different area in program or data memory. Likewise, portions of a section can be spread through several different modules and automatically combined into a single area by the linker.

Sections provide the programmer with complete control over the memory mapping of a program without requiring absolute addressing. A module can contain some relocatable sections and some absolute sections, but a single section is either entirely absolute or entirely relocatable. Section 3.4.2 describes section definition in more detail.

CHAPTER 2
ASSEMBLY LANGUAGE SYNTAX

2.1 INTRODUCTION

The basic component of an asmS8 program is the assembly language statement. An assembly language statement can be up to 128 characters in length and is terminated by an end-of-line character. A statement can include four fields:

- Statement labels
- An opcode
- Operands
- Comments

A typical asmS8 statement might look like:

```
LABEL1: LD R2,R5  ;comment
```

where LABEL1 is the statement label (signified by the colon), LD is the opcode, R2 is the destination operand, R5 is the source operand, and a comment is indicated by a semicolon. For compatibility with Zilog's Z8000 assembler, comments can begin with //, although this assembles slower.

All fields are optional; label and comment fields can start in any column; the opcode and operands cannot start in column 1. The statement can have zero or more operands, depending on the opcode selected. The following sections describe conventions that must be observed in writing a program statement.

2.2 SYMBOLIC NOTATION

Symbolic identifiers can include opcodes, pseudo-ops, special symbols, and labels. Legal identifiers can be up to 127 characters in length, and consist of one or more alphabetic characters, digits, or the characters: comma (,), dollar sign ($), question mark (?), period (.), at sign (@), or single quote mark ('). Upper and lower case letters are considered unique, and all characters are significant.

The only restriction on symbols is that they cannot start with a digit or single quote mark ('). Since some older programs can rely on having only the first eight characters of a symbol being significant, a global variable called $'SYMLEN is provided to limit the number of significant characters in a symbol. Appendix B describes global variables in more detail.

2.2.1 Labels

Any statement that is referenced by another statement must be labeled, and any statement can contain one or more labels. A label is a symbolic identifier that can represent:

- An address (up to 16 bits)
- An I/O port
- A floating-point number
- Other quantities with up to 80 bits of significance.

When a label is being defined, it can start in any column when immediately followed by a colon (:). If a colon is not used, the label must start in column 1. More than one label can be defined on the same line, for example:

```
LABEL1: LABEL2: ... LABELn: statement
```

A GLOBAL label can be declared by placing two colons after the label on the line where it is defined (e.g., LABEL1::). An EXTERNAL label can be declared by two pound signs that immediately follow (e.g., LABEL2##). A tilde (~) as the first character of a label makes it local to a block, as defined by the .BEGIN and .END pseudo-ops.

A label definition preceded by a colon (:LABEL1) specifies that the data type of the label will be the same as the type generated by the rest of the statement. These labels can be checked across module boundaries.

Labels for registers are given special treatment. Indexing is the only valid operation. Table 2-1 lists the Z8 System and Control register names. Table 2-2 lists the Super8 system register names and Table 2-3 lists the Super8 Mode and Control register names.

The names of opcodes can be used freely as labels in the same assembly language statements. The assembler can recognize when a string is being used as an opcode rather than as a label.

Table 2-1. Z8 System and Control Registers

Decimal Address	Hexadecimal Address	Register name	Identifier
255	FF	Stack Pointer (bits 7-0)	SPL
254	FE	Stack Pointer (bits 15-8)	SPH
253	FD	Register Pointer	RP
252	FC	Program Control Flags	FLAGS
251	FB	Interrupt Mask Register	IMR
250	FA	Interrupt Request Register	IRQ
249	F9	Interrupt Priority Register	IPR
248	F8	Ports 0-1 Mode	P01M
247	F7	Port 3 Mode	P3M
246	F6	Port 2 Mode	P2M
245	F5	T0 Prescaler	PRE0
244	F4	Timer/Counter 1	T0
243	F3	T1 Prescaler	PRE1
242	F2	Timer/Counter 1	T1
241	F1	Timer Mode	TMR
240	F0	Serial I/O	SIO
127-4	7F-04	General-purpose registers	
3	03	Port 3	P3
2	02	Port 2	P2
1	01	Port 1	P1
0	00	Port 0	P0

Table 2-2. Super8 System Registers

Decimal Address	Hexadecimal Address	Register name	Identifier
222	DE	System mode	SYM
221	DD	Interrupt Mask Register	IMR
220	DC	Interrupt Request Register	IRQ
219	DB	Instruction Pointer (Bits 7-0)	IPL
218	DA	Instruction Pointer (Bits 15-8)	IPH
217	D9	Stack Pointer (Bits 7-0)	SPL
216	D8	Stack Pointer (Bits 15-8)	SPH
215	D7	Register Pointer 1	RP1
214	D6	Register Pointer 0	RP0
213	D5	Program Control Flags	Flags
212	D4	Port 4	P4
211	D3	Port 3	P3
210	D2	Port 2	P2
209	D1	Port 1	P1
208	D0	Port 0	P0

Table 2-3. Super8 Mode and Control Registers

Decimal Address	Hexadecimal Address	Bank Number	Register Name	Identifier
255	FF	0	Interrupt Priority	IPR
		1	Wake-up Mask	WUMSK
254	FE	0	External Memory Timing	EMT
		1	Wake-Up Match	WUMCH
253	FD	0	Port 2/3B Interrupt Pending	P2BIP
252	FC	0	Port 2/3A Interrupt Pending	P2AIP
251	PB	0	Port 2/3D Mode	P2DM
		1	UART Mode B	UMB
250	FA	0	Port 2/3C Mode	P2CM
		1	UART Mode A	UMA
249	F9	0	Port 2/3B Mode	P2BM
		1	UART Baud Generator (bits 7-0)	UBGL
248	F8	0	Port 2/3A Mode	P2AM
		1	UART Baud Generator (bits 15-8)	UBGH
247	F7	0	Port 4 Open Drain	P4OD
246	F6	0	Port 4 Direction	P4D
245	F5	0	Handshake 1 Control	H1C
244	F4	0	Handshake 0 Control	H0C
241	F1	0	Port Mode	PM
		1	DMA Count (bits 7-0)	DCL
240	F0	0	Port 0 Mode	P0M
		1	DMA Count (bits 15-8)	DCH
239	EF	0	UART Data	UIO
237	ED	0	UART Interrupt Enable	UIE
236	EC	0	UART Receive Control	URC
235	EB	0	UART Transmit Control	UTC
229	E5	0	CTR 1 Capture (bits 7-0)	C1CL
		1	CTR 1 Time Constant (bits 7-0)	C1TCL
228	E4	0	CTR 1 Capture (bits 15-8)	C1CH
		1	CTR 1 Time Constant (bits 15-8)	C1TCH
227	E3	0	CTR 0 Capture (bits 7-0)	C0CL
		1	CTR 0 Time Constant (bits 7-0)	C0TCL
226	E2	0	CTR 0 Capture (bits 15-8)	C0CH
		1	CTR 0 Time Constant (bits 15-8)	C0TCH
225	E1	0	CTR 1 Control	C1CT
		1	CTR 1 Mode	C1M
224	E0	0	CTR 0 Control	C0CT
		1	CTR 0 Mode	C0M

2.2.2 Condition Codes

Condition codes are recognized only as operands of instructions that take them. For example, the statement

```
JR Z, Label
```

causes Z to be treated as the condition code for zero.

The condition codes and flag settings they represent are listed in Table 2-4.

Table 2-4. Z8 and Super8 Condition Codes

Binary	Mnemonic	Meaning	Flags Set
0000	F	Always False	-
1000	T	Always True	-
0111	C	Carry	C=1
1111	NC	No Carry	C=0
0110	Z	Zero	Z=1
1110	NZ	Not Zero	Z=0
1101	PL	Plus	S=0
0101	MI	Minus	S=1
0100	OV	Overflow	V=1
1100	NOV	No Overflow	V=0
0110	EQ	Equal	Z=1
1110	NE	Not Equal	Z=0
1001	GE	Greater than or equal	(S XOR V) = 0
0001	LT	Less than	(S XOR V) = 1
1010	GT	Greater than	(Z OR (S XOR V)) = 0
0010	LE	Less than or equal	(Z OR (S XOR V)) = 1
1111	UGE	Unsigned greater than or equal	C=0
0111	ULT	Unsigned less than	C=1
1011	UGT	Unsigned greater than	(C = 0 AND Z = 0) = 1
0011	ULE	Unsigned less than or equal	(C OR Z) = 1

2.3 OPERATIONS AND OPERANDS

An operation is a mnemonic that represents an instruction.

The assembler also supports a restricted mode that handles only Z8 instructions.

An operation in a program statement can be followed by one or more operands, which are general expressions separated by spaces or commas. Macro parameter lists are the only exceptions since they require parameters to be separated by commas only. Commas do not have the same effect as spaces because two commas in a row denote an omitted operand. A carriage return always serves as a statement delimiter. No more than one statement can be on single line, and a single statement cannot span more than one line.

An operand in a program statement can be:

- Data to be processed (immediate data)
- The designation of a location from which data is to be taken (source address)
- The designation of a location where data is to be placed (destination address)
- The address of a program location to which program control is to be passed
- A condition code, used to direct program flow

Although there are a number of valid combinations of operands, there is one basic convention to remember: the destination operand always precedes the source operand(s). Refer to the specific instructions in the appropriate (Super8 or Z8) Technical Manual for valid operand combinations, and for information about addressing modes.

2.4 COMMENTS

A comment is any string of characters following a semicolon (;) or two slashes (//) in a statement line. Comments have no functional effect on the assembly of a program--they are used only for documentation.

Comments can start in any column of a line, and a statement can consist of only a comment. Comments terminate at the end of the line.

2.5 ARITHMETIC EXPRESSIONS

The asmS8 assembler has a rich set of operators and expressions to handle arithmetic operations. This section first deals with specific formats for arithmetic statements, then follows with a discussion of constants and special symbols.

2.6 EXPRESSIONS AND OPERATORS

Arithmetic expressions can be as long as 80 bits, and are examined from left to right. Precedence (order of evaluation) is as follows:

- Operators and operands are accumulated. As soon as an operator is found that has a precedence level greater than or equal to the last operator encountered, all lower-precedence operations up to the new operator are performed.

- First prefix operations are performed, from right to left (inside out), then postfix and infix operations are performed from left to right.

- Operands (labels and subexpressions in parentheses) are considered to be of precedence level 0.

The operators and their precedence (order of evaluation) are given in Table 2-5. The character "-" after the precedence means that the operation is not present in the Z8 assembler. The last column gives the PLZ/ASM equivalent, if there is one.

II

Table 2-5. Operations and Precedence

Operator	Function	Precedence	PLZ/ASM
operand			
label		0	label
constant		0	
constant			
(...)	Grouping	0	(...)
prefix			
@	Register indirect	1	@
~	Declare local symbol	1	
postfix			
##	Declare external	1	
prefix			
^HB	High byte	2-	
^LB	Low byte	2-	
^HW	High word	2-	
^LW	Low word	2-	
+	Unary plus	2	+
-	Unary minus	2	-
^C	1's complement	2	LNOT
^B	Binary-coded decimal	2	
^BYTE	Byte (8 bit)	2-	
^WORD	Word (16 bit)	2-	
^LONG	Long (32 bit)	2-	
^QUAD	Quad (64 bit)	2-	
^QUINT	Quint (80 bit)	2-	
^ADDR	Address (16 bit)	2-	
^REV	Byte reverse	2-	
^FWD	Forward reference	2-	
^EXT	External reference	2-	
infix			
**	Exponentiation	3-	
*	Multiplication	4	*
/	Division	4	/
^MOD	Modulo	4-	MOD
^<	Shift right	4	SHL
^>	Shift left	4	SHR

Table 2-5. Operations and Precedence (Continued)

Operator	Function	Precedence	PLZ/ASM
+	Addition	5	+
-	Subtraction	5	-
^CAT	String concatenation	5-	
^$ or ^&	Bitwise AND	6	LAND
^|	Bitwise OR	7	LOR
^X	Bitwise exclusive OR	7	LXOR
=	Equal	8-	
>	Greater than	8-	
<	Less than	8-	
>=	Greater than or equal	8-	
<=	Less than or equal	8-	
^UGT	Unsigned >	8-	
^ULT	Unsigned <	8-	
<>	Not equal	8-	
^SEQ	Strings equal	8-	
^SNE	Strings not equal	8-	
prefix			
!	Not-zero	9-	
infix			
&&	Logical AND	9-	
||	Logical OR	10-	
prefix			
#	Immediate operand	11	
#			
postfix			
adr[...]	Indexing	11-	
adr(...)	Indexing	11	a()

Arithmetic is NOT DEFINED on floating-point values.

The result of a test is zero if false, and all ones if true. For purposes of conditional assembly and logical operations, non-zero is considered to be TRUE.

Parentheses can be used for grouping as well as to alter the predecedence of evaluation.

Indexing (parentheses or square brackets) can be applied to strings to extract a particular character, or to addresses or offsets to denote indexed addressing.

The type operators (like ^BYTE) can be used to tell the assembler that a forward or external reference will fit in a given size.

The ^FWD and ^EXT operations return 1 if the value of their operand is forward-referenced or external; they otherwise return 0.

There are no restrictions on the relocation modes of integer operands, since the linker can support arbitrary integer arithmetic on relocatable and external symbols. However, operations on strings cannot be passed to the linker.

Some expression operators consist of multiple characters. There are three main forms, as shown in Table 2-6.

Table 2-6. Expression Operators

Form	Example	Description
??	<=	Two punctuation characters
^?	^<	"^" plus single punctuation character
^x	^FS	"^" plus any number of letters
id		An identifier

No identifiers are used as expression operators in the assembler as supplied. However, the user can define them to achieve compatibility with PLZ/ASM and other assemblers.

2.7 CONSTANTS

A constant value is one that doesn't change throughout the program. Constants can be expressed as numbers (integer and floating-point), character sequences, or as symbolic names representing a constant value. Constants supported by the assembler include integers, floating-point numbers, characters, and character strings.

Integers start with a digit (leading zero is sufficient) and can contain a base indicator:

B	Binary
D, E or e	Decimal
H or X	Hexadecimal
O or Q	Octal

This is an extension that was made to allow C-style constants. Base indicators and hexadecimal digits can be in any mixture of upper and lower case. The default value is decimal.

In addition, the PLZ base-tag convention is supported:

%	Hexadecimal
%(8)	Octal
%(2)	Binary

Floating-point numbers start with a digit and contain a decimal point. They can optionally contain the letter E or e followed by an optional sign and an exponent. Floating-point numbers are always in base 10.

Characters and character strings are enclosed in single or double quotes. If an escape character is defined, C-type escape sequences are permitted. The escape character is the value of the special symbol $'STRESC. The characters permitted after the escape character and their meanings are noted in Table 2-7.

Table 2-7. Escape Characters

Permitted Characters	Meaning
q	The string's quote character
n	Newline (line feed)
r	Carriage return
f	Form feed
t	Tab
b	Backspace
'	Single quote
"	Double quote
\	Backslash
%dd	(2 hex digits)--arbitrary character
ddd	(1-3 octal digits)--arbitrary character

The number base of the digits form of escape is given by $'SBASE (default 8).

2.8 LOCATION COUNTER

The symbols ($) or (.) refer to the current value of the location counter (corresponding to the address where the first byte generated by the statement is loaded). Either one of these symbols can be used as an operand in any arithmetic expression (but their use does not imply the use of PC-relative addressing). The arithmetic expression is computed at assembly or link time.

CHAPTER 3
PSEUDO-OPS

3.1 INTRODUCTION

The asmS8 assembler permits the use of pseudo-ops (pseudo operation codes). These pseudo-ops do not cause the assembler to generate object code, but rather specify actions to be taken by the assembler. Pseudo-ops use the same line format as standard instructions (label, opcode, operands, comments). Pseudo-ops can begin in any column except column 1. The pseudo-ops permitted by the asmS8 assembler are grouped by function and are described in the following sections. Table 3-1 lists the pseudo-op abbreviations and their meanings.

II

Table 3-1. Pseudo-Op Description Abbreviations

Abbreviations	Meaning
n	Numeric expression
s	String
sn	String or numeric expression
d	Decimal digit
p	Actual parameter (see note 1)
f	Formal parameter (see note 2)
l	Optional label, more than one Permitted
ll	Required label, one only
?	Optional

Notes for Table 3-1:

1. An actual parameter is a string enclosed by macro quotes (normally {...}) or any sequence of characters delimited by a comma, space (if $'BSEP is set), end-of-line, or semicolon. (Refer to $'MACBEG and $'MACEND in section 4.2.2).

2. A formal parameter is either a label or an actual parameter that does not start with a character that can denote a label.

3.2 RELOCATION PSEUDO-OPS

The following pseudo-ops are used to specify the relocation of code within memory.

3.2.1 Origin

General Form:

```
1 .ORG  n
```

Description:

The .ORG pseudo-op sets the location counter to the value of the expression n. In specifying where the object code is located, the location counter serves the same function for the assembler as the Program Counter does for the CPU.

The location counter is set to the value of the expression, so that the next machine instruction or data item will be located at the specified address. The expression must not contain any forward references, but can be relocatable. The location counter is initially set to zero, so if no .ORG statement precedes the first instruction or data byte in a section, that byte will be assembled at location zero (relative to the start of the section). Any label that is present will be assigned the same value as the expression. A module can contain any number of .ORG statements.

The mode of the expression in an .ORG pseudo-op cannot be external and depends on the relocatability of the section. If a section is absolute, the .ORG pseudo-op serves to assign an absolute address to both the location counter and the label. In addition, any .ORG statement will also set the starting address of an absolute section when it immediately follows the .SECTION statement.

In a relocatable section, the expression will be treated as any offset relative to the origin of the module. Thus the label on an .ORG statement in a relocatable module will have a relocatable mode. For example, the effect of the statement

```
Label   Opcode   Operand
LAB:    .ORG     100
```

within a relocatable section would be to set the location counter to the beginning of the section plus 100, assign the label LAB the value 100, and make that label relocatable. A simply relocatable expression in an .ORG can be used to change to another section.

Relocatable sections do not generally contain .ORG statements, since the pseudo-op is useful only to reserve space within the module (in a manner similar to the .DEFS pseudo-op).

Example:

```
START1: .ORG %10 ;Start section 1 at the hex
                 ;address 10
```

3.2.2 Phase

General Form:

```
.PHASE   n
```

Description:

The .PHASE pseudo-op assembles the code that follows to execute starting at address n. Labels will be defined as if an origin pseudo-op (.ORG) had been issued, but the address into that code is not affected. This pseudo-op is provided for pieces of code that are going to be moved (for example, from ROM to RAM) before they are executed.

Example:

```
.PHASE  500
```

3.2.3 Dephase

General Form:

```
.DEPHASE
```

Description:

The .DEPHASE pseudo-op terminates the effect of a preceding .PHASE pseudo-op.

Example:

```
.DEPHASE
```

3.3 LABEL DEFINITION PSEUDO-OPS

Labels on instructions are automatically assigned the current value of the location counter. The pseudo-ops .EQU and .SET can be used to assign arbitrary values to symbols. To facilitate inter-module communication, certain symbols can be declared to be either .GLOBAL or .EXTERNAL to a particular module. .EQU and .SET require that the expression have no forward references (it can contain previously declared external symbols).

3.3.1 Equate

General Form:

```
ll  .EQU  n
ll   =    n
```

Description:

The .EQU pseudo-op assigns the value of the expression n to the symbol in the label field ll. The label cannot be redefined in this source program. The expression can include a register or other addressing mode.

Using symbolic names for constant values in place of numbers enhances the readability of a program and tends to make the code self-documenting. For instance, the symbol BUFLEN is a more descriptive name for a value than just the number 72. Furthermore, if a value that is used throughout a program needs to be changed, the .EQU statement can simply be modified rather than finding all occurrences of the number 72.

Example:

```
TWO   .EQU 2     ;the symbol TWO now has
                 ;a value of 2
```

3.3.2 Set Re-definable Label

General Form:

```
ll .SET n
```

Description:

This pseudo-op assigns the value of the expression n to the symbol in the label field ll. The label assignment can be changed using a subsequent .SET pseudo-op. The .SET pseudo-op is identical to the .EQU pseudo-op except that the assigned label can appear in multiple .SET pseudo-ops in the same program.

In general, use the .EQU for symbol definition since the assembler will generate error messages for multiply-defined symbols. This can indicate spelling errors or some other oversight by the user. .SET should be reserved for special cases where the same symbol is re-used (e.g., in conjunction with the assembly of macros).

.EQU and .SET require that the expression have no forward references (it can have external symbols provided they have been declared previously).

Example:

```
COND1  .SET   150           ;set initial value to 150
COND1  .SET   COND1 + 100   ;increment value by 100
```

3.3.3 Define Arbitrary Symbol

General Form:

```
ll  .DEF  l
```

Description:

This pseudo-op defines the label ll as an exact synonym for the operand symbol l. Neither the label nor the operand needs to be an identifier; they may be punctuation characters such as + . If the label is non-alphabetic, it must be preceded by a colon.

Example:

```
AND   .DEF  ^&

STORE .DEF  LD

:|    .DEF  ^|
```

3.3.4 Global

General Form:

```
.GLOBAL ll1,...lln
```

Description:

These pseudo-ops specify that each of their operands are symbols that are defined in the current module and that the name and value of each operand is made available to other modules that contain an .EXTERN declaration for any symbol. There can be one or more names separated by commas (or no names at all). .GLOBAL pseudo-ops can occur anywhere within the source text.

Example:

```
.GLOBAL   ENTRYA, EXITA, ENTRYB, EXITB
```

3.3.5 External

General Form:

```
.EXTERN ll1,...lln
```

Description:

This pseudo-op specifies that each of the operands are symbols that are defined in some other module, but are referenced in the current module. The syntax is the same as .GLOBAL.

3.3.5 External (Continued)

.EXTERN pseudo-ops can occur anywhere within the source text. The .EXTERN pseudo-op assigns each name an external mode, which allows the name to be used in arbitrary expressions elsewhere in the module, subject to the rules for external expressions. If an .EXTERN and a .GLOBAL definition for the same label appear in the same module, the .GLOBAL pseudo-op will take precedence.

An external symbol can be assigned a value using either a .SET or .EQU pseudo-op. An assigned value will be the default value of a symbol if it is not resolved when the object module is linked.

Example:

```
.EXTERN ENTRYA, EXITA, ENTRYB, EXITB
```

3.4 MODULE AND SECTION PSEUDO-OPS

The following pseudo-ops are used to name the object module, and to define specific areas of source code that can be relocated separately.

3.4.1 Module Definition

General Form:

```
.MODULE p p?
```

Description:

This pseudo-op defines the name of the module. If given, the second parameter becomes the target name in the object module. Otherwise, the target name will be "Z8" or "ZS8". The target name is a universal object file format field name for use by other programs such as a loader (see the Universal Object File Utilities User's Guide).

There can be only one .MODULE statement in a module. If no .MODULE statement is given, the module takes the name of the source file with its extension (.s) deleted.

Example:

```
.MODULE Main ;Define main module
```

3.4.2 Section Definition

General Form:

```
1 .SECTION 11,...ln
1 .PSEC    11,...ln
```

Description:

These pseudo-ops start a section. The first parameter is the name of the section, and can be null when terminated by a comma. Any other parameters are the universal object file attributes of the section (see the Universal Object Files Utilities User's Guide). When given, a statement label is defined as a pseudo-op that will direct assembly output to that section. Assembly can also be directed to the section by giving another .SECTION command with the same section name.

The following section changing operations are predefined:

Name	**Meaning**
.DATA	Data section
.CODE	Code section
.BSS	BSS section
.ABS	Absolute section
.CSEC	Common section

All of these direct assembly to a section with the same name and appropriate attributes. The default section is a nameless and relocatable section; to return to the default section, use a .SECTION command with no parameters.

The following operations enclose blocks of local symbols:

Name	**Meaning**
.BEGIN	Begin local symbol block
{	Begin local symbol block
.END	End local symbol block
}	End local symbol block

Local symbols are defined with a tilde character "~" at the beginning. .BEGIN and { are synonymous, as are .END and }. Furthermore, blocks can be nested.

Example:

```
   .BEGIN
L1:
   .BEGIN
L1:
   .END
   .END
```

3.4.2 Section Definition (Continued)

Note that a .END without a matching .BEGIN will mark the end of the source program (see section 3.7.1).

3.5 GENERAL DATA DEFINITION OPERATION

Pseudo-ops are provided to define message, text, character string, and data size.

3.5.1 Data Definition

General Form:

```
1 .DD sn1,...,snn
        or
1 .DD repeat-count(data)
```

Description:

This pseudo-op assembles a list of data items. Any number of expressions or strings can be listed in a .DD statement. Each item listed is stored in its natural length: expressions involving addresses or forward references are stored in 16-bit words, expressions with values less than 256 are stored in one 8-bit byte, and strings are stored "as is."

Strings that are not used as numbers (no arithmetic operators are applied to them) are not affected by special symbols $'STRLEN and $'STRORD. Operators like .BYTE can be used to force an expression to an appropriate length.

Example:

```
DATA: .DD ZED+100
      .DD "This is a string"
```

3.5.2 Sized Data Definition

General Form:

```
1 .BYTE   n1,...,nn
1 .WORD   n1,...,nn
1 .LONG   n1,...,nn
1 .QUAD   n1,...,nn
1 .QUINT  n1,...,nn
1 .EXTEND n1,...,nn
```

Description:

These pseudo-ops define data of a specified size. Any number of expressions can be listed provided each fits within the specified data size. These pseudo-ops take each operand and generate object code of the size specified, locating the most significant byte at the current value of the location counter, and the next most significant byte at the next higher location.

The mode of the expression can be either absolute, relocatable, or external. If present, a label will be assigned the address of the first data item. String arguments are always subject to the processing specified by $'SnLEN and $'SnORD (i.e., converted to numbers).

Example:

```
WORDS: .WORD 512,ABLE
```

3.5.3 Define ASCII String

General Form:

```
1 .ASCII sn1,...,snn
```

Description:

This pseudo-op defines message strings or byte data. A parameter can be either an expression or a string. Any number of parameters can be listed. An expression must fit into a single byte area; strings are stored completely.

Examples:

```
Label   Opcode          Operand

MSG:    .ASCII        'HELLO THERE', x+1
```

3.5.4 Define ASCII String with Length

General Form:

```
1  .ASCIL  s1,...,sn
```

Description:

This pseudo-op defines strings, with each string preceded by a byte containing its length. Parameters can also be expressions, each of which is also stored with a byte containing its length.

Example:

```
TXT: .ASCIL 'OPEN','CLOSE'
```

3.5.5 Define ASCII String with Flagged Last Character

General Form:

```
1 .ASCIC   s1,...,sn
```

This pseudo-op defines character strings. The high-order bit of the last character of each string is set to one (1); the high-order bits of all other characters in the string are set to zero (0).

3.3.5 Define ASCII String with Flagged Last Character (Continued)

Example:

```
CHARS: .ASCIC  'ABCD','EFGH'
```

3.5.6 Define Null-Terminated ASCII Strings

General Form:

```
1  .ASCIZ s1,...,sn
```

Description:

This pseudo-op defines character strings with an additional zero (null) byte at the end of each string.

Example:

```
label:  ASCIZ 'S1',S2
```

3.5.7 Reserve Space

General Form:

```
1 .BLKB  n    Reserve a block of bytes
1 .BLKW  n    Reserve a block of words
1 .BLKL  n    Reserve a block of longwords
1 .BLKQ  n    Reserve a block of quadwords
1 .BLKX  n    Reserve a block of extended words
```

Description:

These pseudo-ops reserve space in differing word lengths. The operand n specifies the number of words to be reserved for data storage starting at the current value of the location counter. Except for .BLKB, these pseudo-ops are aligned on word boundaries. When present, a label will be assigned the address of the first byte reserved.

The expression can evaluate to any quantity; however, the mode must be absolute and not have forward references. Any symbol appearing in the expression must have been defined before the assembler encounters the expression.

Example:

```
label: .BLKW 5
```

3.5.8 General Reserve Block

General Form:

```
1 .BLOCK n, n?
```

Description:

This pseudo-op reserves n bytes of space in memory. One operand (n) specifies the number of bytes to be reserved for data storage starting at the current value of the location counter. When provided, the second operand is the alignment boundary for the block. Any label will be assigned the address of the first reserved byte.

The expression can evaluate to any quantity, but the mode must be absolute and not have forward references. Any symbol appearing in the expression must be defined before the assembler encounters the expression.

This pseudo-op reserves storage by incrementing the location counter by the value of the first expression. Since no object code is generated into the storage area, the contents of storage during initial program execution are unpredictable.

Example:

```
STORE: .BLOCK 512
```

3.5.9 Alignment

General Form:

```
 .ALIGN n?
```

Description:

This pseudo-op aligns the next item on a multiple of n bytes. If the next statement is a .SECTION pseudo-op, the start of the section is aligned. If the parameter n is omitted, a word alignment default value of 2 is assumed.

Example:

```
FORMAT: .ALIGN 4
```

3.5.10 Even or Odd Alignment

General Form:

```
.EVEN
.ODD
```

Description:

These pseudo-ops align the next item on an even or odd boundary.

3.6 CONDITIONAL ASSEMBLY PSEUDO-OPS

Conditional assembly permits the programmer to inhibit or enable the assembly of defined portions of the source code depending on the presence of a predetermined condition.

General Form:

- Start Conditional Block
  .IF n
- Separate True and False Conditional Blocks
  .ELSE
- End Conditional Block
  .ENDIF

Description:

.IF defines the start of the conditional code block and tests for the true (non-zero) or false (zero) state of the expression n. .ELSE separates the code that is assembled if the expression is true from the code that is assembled if the condition is false (.ELSE is optional). .ENDIF defines the end of the conditional code block. Conditional blocks can be nested up to 80 deep.

The mode of the expression can be either relocatable or absolute. Forward or external expressions generate a warning, and are always considered to be true.

Notice that the definition of symbols within a conditional assembly block can be inhibited, and thus references to these symbols elsewhere in the module can cause undefined symbol errors. In particular, the label on an .ELSE pseudo-op is part of the true block, and will not be defined if the assembly is inhibited on that portion of the program.

Conditional assembly is particularly useful when a program needs to contain similar code sequences for slightly different applications. Rather than generating a multitude of programs to handle these situations, the application-dependent sections of code can be enclosed by the conditional pseudo-ops within a single program. Thus, the user generates different object modules from subsequent assemblies of the same source by changing the values of several symbols used to control the conditional assembly.

Another common use of conditional assembly is in conjunction with macros to control generation of code dependent on the value of parameters (see Chapter 4).

Example:

```
IF FLAG
    .
    .       ;assembled if FLAG non-zero
    .
.ELSE
    .
    .       ;assembled if FLAG equals zero
    .
.ENDIF
```

3.7 ASSEMBLER CONTROL PSEUDO-OPS

Pseudo-ops are provided to: control the format of printed listings, control the information presented on the listings, control the printing of errors or warning messages, and to establish the compatibility mode of the assembler.

3.7.1 End Program

General Form:

```
1 .END n?
```

Description:

This pseudo-op specifies the end of source code. Any expression is taken as the starting address of the program. The .END pseudo-op signifies the end of the source program, and thus any subsequent text will be ignored and will not appear in a listing.

Any label will be assigned the current value of the location counter. Operands are ignored. If a source program does not contain an .END pseudo-op, then the end-of-file mark in the assembler command line will signify the end of the program.

Examples:

```
EXIT: .END ;end of module

.END START
```

3.7.2 Include

General Form:

```
.INCLUDE  p
```

Description:

This pseudo-op includes the source file identified by the parameter (p) into the source stream immediately following this statement. The usual use of this statement is to include items such as files of macro definitions, lists of .EXTERNAL declarations, lists of

3.7.2 Include (Continued)

.EQU statements, or commonly used subroutines into the source stream. However, this pseudo-op can be used anywhere in a program. The file name given must follow the normal convention for specifying source file names.

.INCLUDE pseudo-ops can be used in files specified in a preceding .INCLUDE pseudo-op. These pseudo-ops can be nested to a depth of four deep. If the .INCLUDE pseudo-op appears within a macro definition, the file will be included every time the macro is expanded. .INCLUDE pseudo-ops can be used in conditionals.

Example:

```
.INCLUDE FILE1
```

3.7.3 Page Title

General Form:X

```
.TITLE p1,...pn
```

Description:

This pseudo-op causes the string specified in parameters to be printed at the top of each page of the listing.

Example:

```
.TITLE  Program for Interest Calculation
```

3.7.4 Page Subtitle

General Form:

```
.SUBTTL p1,...pn
```

Description:

This pseudo-op prints strings specified in parameters on the second line of following pages in the listing. The subtitle on the first page of the listing will be the name of the source file. An outer layer of quotes will be ignored.

Example:

```
.SUBTTL   Created by P. Jones
```

3.7.5 Listing Control

General Form:

- Control Listing

  .LIST p

- Control Warning Listing

  .WLIST p

- Control Conditional Listing

  .CLIST p

- Control Macro Listing

  .MLIST p

- Control Macro Object Listing

  .XLIST p

Description:

These pseudo-ops cause an output listing file to be generated according to the pseudo-op(s) used and the parameter given.

The parameters given for each of the listing control pseudo-ops can be any one of the following symbols:

Value	Meaning
ON	Include in listing file.
OFF	Do not include in listing file.
PUSH	Save current value of pseudo-op control status in appropriate variable.
POP	Restore saved value of pseudo-op control status from appropriate variable.

The variables $'LIST, $'WLIST, $'CLIST, $'MLIST, and $'XLIST are used as 80-bit pushdown stacks to store and recover the current state of the parameter given in their respective list control pseudo-op. The parameter state value is stored in the low-order bits of the variable.

Pseudo-op .LIST with p=ON enables a listing file of the source to be generated. When P=OFF, .LIST prevents a listing file from being generated.

3.7.5 Listing Control (Continued)

Pseudo-op .WLIST with p=ON enables warning messages to be included in the listing file. When p=OFF, .WLIST prevents warning messages from being included in the listing file.

Pseudo-op .CLIST with p=ON enables those portions of the source file that are conditionally skipped to be included in the listing file. When p=OFF, .CLIST prevents those "conditionally skipped" portions of the source file from being included in the listing file.

Pseudo-op .MLIST with p=ON enables the expansion of macros to be included in the listing file. When p=OFF, .MLIST prevents macro expansions from being included in the listing file.

Pseudo-op .XLIST with p=ON enables the listing of binary object code to be included in the listing file. When p=OFF, .XLIST prevents these extra binary object lines from being included in the listing file.

The default value for all listing control listings is p=ON.

Example:

```
.LIST ON
```

3.7.6 List Error Message

General Form:

```
.ERROR  s
```

Description:

This pseudo-op causes the message given in string (s) to be generated and sent to the terminal and the listing.

Example:

```
.ERROR  'SYNTAX ERROR'
```

3.7.7 List Warning Message

General Form:

```
.WARN  s
```

Description:

This pseudo-op causes the warning message given in string (s) to be generated and sent to the standard output.

Example:

```
.WARN  'POSSIBLE PROBLEM HERE'
```

3.7.8 Start New Page

General Form

```
.PAGE  n
```

Description:

This pseudo-op causes the listing to be paginated. The page size is set at the value given in n. If n is zero, the assembler will not paginate the listing. Page size is given in number of lines per page.

The default action is not to paginate the listing, since system utilities can be used for that purpose. .PAGE with no operand simply starts a new page in the listing, and is equivalent to a line containing a form feed.

Example:

```
.PAGE  66   ;set page size to 66 lines
```

3.7.9 Search Library

General Form:

```
1  .LIBRARY  p l1,...ln?
```

Description:

This pseudo-op puts a directive into the object file that instructs the linker to search a given library file (the first parameter) for the definitions of external symbols. If labels are given in the parameter(s), the library is searched only for those external labels.

Example:

```
.LIBRARY clib.a Subr1, Subr2, Subr3

.LIBRARY xyzlib
```

3.7.10 Object File Comment

General Form

```
.OCOMMENT  n? s
```

Description:

This pseudo-op enters the text given in string (s) into the object file listing as a comment. Any value given for n is used as the "comment level" value. Comments below a link-time settable level will appear in load maps.

Example:

```
 .OCOMMENT    3,'tables start here'
```

CHAPTER 4
MACROS

4.1 GENERAL DESCRIPTION

Macros provide a means for users to define their own opcodes or to redefine existing opcodes. A macro is a portion of a program invoked by its name. It begins and ends with pseudo-ops, and can contain any assembler constructs, including pseudo-ops and macros. Two types of macros can be used in asmS8 programs: MACROs and PROCs.

MACROs are the familiar string substitution macros used in other assemblers. Parameter strings provided in the macro's invocation are substituted in the body of the macro. MACRO parameters must be separated by commas, and can contain blanks.

PROCs are call-by-value, procedure-type macros. The parameters provided in the invocation statement are expressions, and their values are substituted into the body of the macro. As with ordinary opcodes, PROC parameters can contain blanks either before or after operators. Likewise, commas between expressions are optional.

In general, a macro definition consists of the block of code beginning with a "start" pseudo-op and ending with an "end" pseudo-op. The statement containing the start pseudo-op requires a label. It serves as the name of the macro, and is used to invoke it. Each statement between the start and end statements is stored in the assembler's symbol table as the definition of the macro. These statements can include macro invocations and definitions. In addition, recursion is allowed.

The statements of the macro body are not assembled at definition time. As a result, they do not define labels, generate code, or cause errors until the macro is invoked. Macros must be defined before they are invoked.

A macro is invoked by using its name as an opcode at any point after the definition. Every macro definition has an implicit parameter named #$YM. This can be referenced by the user in the macro body, but should not explicitly appear in the .MACRO statement.

4.1 GENERAL DESCRIPTION (Continued)

At expansion time, each occurrence of #$YM in the definition is replaced by a string representing a 4-digit hexadecimal constant. This string is constant over a given macro expansion. However, it increases by one for each macro invocation to avoid multiple definition errors. This provides unique labels for different expansions of the same parameter.

4.2 MACRO OR STRING MACRO

MACRO is the string substitution macro.

4.2.1 MACRO Definition

The general form of a MACRO definition is:

```
ll .MACRO f1,...,fn  ;start MACRO pseudo-op.
    .
    .    (statements that form body of MACRO)
    .
    .ENDM             ;end MACRO pseudo-op.
```

The required label serves as the name of the MACRO, to be used on invocation. A formal parameter (f1,...,fn) can be either a label or a string of any characters except blanks, commas, or semicolons. Furthermore, parameters must start with a character that cannot start a label. Formal parameters that are labels are recognized in the macro body anywhere a label would be recognized (i.e., labels or opcodes). Parameters that are not labels are recognized anywhere (e.g., within labels, strings, or comments).

Parameters are scanned left to right for a match, so the user is cautioned not to use parameter names that are prefix substrings of later parameter names. Formal parameters are not entered in the symbol table.

MACROs can contain any statements including MACRO definitions and invocations, other assembler directives, and conditional assembly. The pseudo-ops .MACRO and .ENDM specify the beginning and end of a MACRO, respectively.

4.2.2 MACRO Special Symbols

The following special symbols are defined for use with MACROs.

They can be reassigned using .SET pseudo-ops, and can be used as operands anywhere a label could be used.

$'MACEVAL   '%'

Used to replace an expression, used as a macro parameter, with its value.

```
$'MACQUOTE     '!'
```

Used to include the following character in a macro parameter, despite any special meaning it may have.

```
$'MACBEG       '{'
$'MACEND       '}'
```

Beginning and ending macro parameter delimiters. If different, they must be properly nested, or they could cause an escape with $'MACQUOTE.

4.2.3 MACRO Invocation and Expansion

A MACRO is invoked when its name is used as the opcode. The rest of the line is made up of "actual parameters"--strings of characters separated by commas, possibly enclosed in quotes (normally {...}). Quoted parameters can include commas as well.

The actual parameters on the invoking line replace the corresponding formal parameters from the defining line wherever they occur in the body of the macro. If legal, a formal parameter is replaced wherever it occurs as an identifier. If a formal parameter is not a legal identifier, it is matched as a string and is replaced wherever it occurs. The statement is assembled after these substitutions, and the resultant code placed in the program in place of the invoking statement.

4.2.4 MACRO Example

Assuming that the label UPDATE has already been defined, the .MACRO invocation

```
START  UPDATE  46,99,current
```

substitutes the actual parameter strings 46, 99, and "current" for the first, second, and third formal parameters within the body of the MACRO named UPDATE.

4.3 PROC OR PROCEDURE MACRO

The procedure (or .PROC) macro is a call-by-value macro. The major difference between a .MACRO and a .PROC is that the parameters of the procedure-type macro are expressions that are evaluated before the .PROC is expanded.

4.3.1 PROC Definition

The general form of a PROC definition is:

```
l1 .PROC l1,...,ln    ;start PROC pseudo-op
    .
    .  (statements that form body of PROC)
    .
    .ENDP             ;end PROC pseudo-op
```

4.3.1 PROC Definition (Continued)

The required label is the name of the .PROC and is used to invoke it. The pseudo-ops .PROC and .ENDP specify the beginning and end of a PROC-type macro. The formal parameters are labels that are recognized only when they are used in expressions or as statement labels. PROCs can contain any statements including macro definitions and invocations, assembler commands, and conditional assembly.

4.3.2 PROC Invocation and Expansion

When a PROC is invoked, the expression parameters are evaluated and substituted into the body of the PROC as values. Then the PROC is assembled normally and its code is inserted into the program in place of the invocation statement.

4.3.3 PROC Example

For example, assume the following PROC definition:

```
ESTIMATE  .PROC  total,average
     .
     .  (body of PROC)
     .
.ENDP
```

Using this invocation:

```
ESTIMATE  sum+12,sum+12/num
```

would substitute the value of sum+12 for the formal parameter "total", and the value of sum+12/num for "average" in the ESTIMATE PROC. These values would then be used by the assembler in assembling the PROC in the program stream.

4.4 SPECIAL MACRO PSEUDO-OPS

Several special pseudo-ops are provided for use within MACROs. These pseudo-ops can stop macro expansions, define labels for each macro invocation, or provide looping capabilities.

4.4.1 Exit Macro

General Form:

```
.EXITM n?
```

Description:

This pseudo-op stops the expansion of a macro. It can be used in all forms of a macro (MACRO or PROC) to force an early termination of the MACRO's expansion. The exit can be made on a conditional basis.

4.4.2 Define Local Symbols

General Form:

```
.LOCAL l1,...,ln
```

Description:

This pseudo-op defines local symbols within a macro. Each symbol given in the list with this pseudo-op is replaced in the expansion of the MACRO by the symbol "..XXXX" where XXXX represents a hexadecimal number unique for each local symbol in each invocation of the macro. When used, the .LOCAL pseudo-op must immediately follow the defining MACRO or PROC statement.

Example:

```
POWER: .MACRO x
       .LOCAL two,three ;two and three will be assigned
                        ;a unique symbol for each
                        ;invocation of the macro.
```

4.4.3 Repeat

General Form:

```
.REPT n
   .
   .
   .
.ENDM
```

Description:

The block of statements between .REPT and .ENDM is repeated n times. The value of n must be absolute and not include forward references.

Example:

```
.REPT 4
   .
   .
   .
.ENDM
```

4.4.4 Repeat On Parameter List

General Form:

```
.IRP f,s
   .
   .
   .
.ENDM
```

4.4.4 Repeat On Parameter List (Continued)

Description:

The quotes are stripped from the string, and the block of statements between .IRP and .ENDM is repeated, with each parameter in the string s replacing the formal parameter f in the expansion of the contained statement.

Example:

```
.IRP X, "4,8"  ;first 4, then 8, is substituted for
               ;each occurrence of X from here to the
               ;end of the macro.
.ENDM
```

4.4.5 Repeat On Character String

General Form:

```
.IRPC f,s
   .
   .
   .
.ENDM
```

Description:

The block of statements between .IRPC and .ENDM is repeated, with each character in s replacing the formal parameter f in the contained statements.

Example:

```
.IRPC X, "1234567"  ;the characters 1
                    ;through 7 are substituted
                    ;for the seven iterations of this
                    ;macro.

ENDM
```

4.5 SPECIAL MACRO OPERATORS

The following sections discuss operators and symbols that are useful mainly within macro definitions or invocations. These symbols are %, !, { }, ^DEF, and ^NUL. Note that the single-character operators can be redefined by changing the value of the corresponding special symbols.

4.5.1 '%' Operator

The symbol % in front of a label in a macro parameter causes the numeric value of the expression to be converted to a decimal ASCII string and incorporated into the parameter. The symbol % will be recognized within a symbol to construct new symbols. The label's value must be absolute, and may not contain a forward reference.

The special symbol $MACEVAL can be used to change the character used for this function from its initial default of "%".

4.5.2 '!' Operator

The character ! in front of a character in a macro parameter makes that character part of the parameter, even if the character is normally treated specially (e.g., , comma, etc.). The special symbol $MACQUOTE can be used to change the character used for this function from its initial default of "!".

4.5.3 {...}

A macro parameter enclosed in braces will have an outer layer of braces eliminated. The beginning and ending braces are the value of $'MACBEG and $'MACEND, respectively, but can be changed.

Beginning and ending braces must be properly nested. If the beginning and ending characters are the same, they cannot be nested. However, the character itself may be entered by either doubling it (e.g., "") or preceding it with '!'.

4.5.4 ^DEF 1

^DEF followed by a symbol expands to a non-zero value if the symbol has been defined (previous to the current line) or 0 if the symbol has not been defined.

4.5.5 ^NUL

^NUL expands to a non-zero value if it is the last token on a line (not counting a comment), or 0 otherwise. The rest of the line is ignored.

CHAPTER 5
PROGRAM INVOCATION

5.1 ASSEMBLER COMMAND LINES AND OPTIONS

The asmS8 assembler accepts various command line options for assembly, creates a listing, and creates an object file in a universal file format suitable for use by such utilities as a loader (see the Universal Object File Utilities User's Guide).

The assembler is invoked as follows:

```
asmS8 [option . . .] file
```

Valid assembler options are listed in Table 5-1.

Table 5-1. Assembler Options

Option	Meaning
-d	Reserved
-en	Stop after n errors
-l	Produce listing for files in file.1
-o objfile	Specify object file name other than a.out
-ob	Produce object in binary form
-oc	Produce object in character form
-on	Produce object with file and line number in comment level 1
-os	Produce object with source lines in comment level 2
-ow	Produce object with user-generated warnings in comment level 2
-p	Produce listing on standard output
-r	Restrict to Z8 instruction set
-s symfile	Get assembler's symbol table initialization from symfile
-u	Treat undefined symbols as externals
-w	Don't list warnings
-x	Produce cross-reference on file.x

If the -l option is given and the source filename ends in ".s", the listing is produced in filename.l. If the -s flag is not used, the assembler will obtain its symbols from a file on /z/bin/asm* whose name was used to invoke the assembler. Normally, this is /z/bin/lib/asm/asmS8**. The symbol file is an ordinary ASCII source file, and can contain any constructs that do not generate object code. This is used to create custom versions of the assembler.

* for VAX/UNIX it is /usr/local/bin/asm

** for VAX/UNIX it is :/usr/local/bin/asm/asmS8

5.2 LISTING FORMAT

The assembler produces a listing of the source program, along with generated object code. The various fields in the listing format are the heading, the location counter (LOC), the object code (OBJ CODE), the statement number (LINE#), and the source statement (SOURCE). They contain the following:

- The heading is on the first page of the listing and contains the date, time, year, file name, and page number, as well as the column headings LOC, OBJ CODE, LINE#, and SOURCE.

- LOC contains the value of the location counter for statements.

- OBJ CODE contains the generated object code. If a statement does not generate object code, this field is blank. Relocatable values are represented as Rsss+nnnnnnnn where ssss is the section number and nnnnnnnn is the offset within the section. Externals are noted by the letter x, with a capital X representing the first byte. An asterisk (*) notes other link-time expressions that are not simply relocatable.

- LINE# contains the sequence number of each line of the source, starting at 1.

- SOURCE contains the source code including labels, opcodes, operands, and comments.

Appendix E shows a sample listing.

5.3 PROGRAM TERMINATION

The assembler returns an error code of 0 if the program has no errors. Otherwise, the assembler returns an error code of 1 and error messages will appear in the listing. These error messages will also be sent to the terminal with the relevant file and line numbers. If possible, an object file will be created even if errors are present. Appendix D lists the error messages and their explanations.

APPENDIX A
PSEUDO-OP SUMMARY

The following abbreviations apply to the pseudo-op summary:

n	Numeric expression
s	String
sn	String or numeric expression
d	Decimal digit
p	Actual parameter
f	Formal parameter
l	Label (optional, more than one allowed)
ll	Label (required, only one allowed)
...	May be repeated
?	Optional
[...]	Not exactly equivalent (either form acceptable)

Label	**Pseudo-Op**	**Operand**	**Meaning**
Relocation Operations			
l	.ORG	n	Origin
	.PHASE	n	Phase
	.DEPHASE		Dephase
Section Operations			
	.MODULE	p p?	Module name
l	.SECTION	l ...	Define a section
Label Definition Operations			
ll	.EQU	n	Equate
ll	.SET	n	Define a label
	.GLOBAL	ll ...	Global symbols
	.EXTERNAL	ll ...	External symbols
Data Definition Operations			
l	.DD	sn ...	Define data
l	.BYTE	n ...	Define byte data
l	.WORD	n ...	Define word data
l	.LONG	n	Define longword data
l	.QUAD	n ...	Define quadword data
l	.QUINT	n ...	Define 5-byte (extended) data
l	.EXTEND	n ...	Define extended data
l	.ASCII	sn ...	Define ASCII string
l	.ASCIL	s ...	Define ASCII string with length

Label	Pseudo-Op	Operand	Meaning
Data Definition Operations — (Continued)			
l	.ASCIC	s ...	Define ASCII string with flagged last character
l	.ASCIZ	s ...	Define null-terminated ASCII string
Reserve Space Operations			
l	.BLOCK	n n?	Reserve a block with optional alignment
l	.BLKB	n	Reserve a block of bytes
l	.BLKW	n	Reserve a block of words
l	.BLKL	n	Reserve a block of longwords
l	.BLKQ	n	Reserve a block of quadwords
l	.BLKX	n	Reserve a block of extended data
Conditional Assembly			
	.IF	n	Start conditional block
	.ELSE	n	False branch of conditional
	.ENDIF	n	End conditional block
Assembler Control Operations			
	.END	n?	End program
	.INCLUDE	p	Include a source file
	.TITLE	p ...	Listing title
	.SUBTTL	p ...	Subtitle
	.LIST	p	Control listing
	.WLIST	p	Control conditional listing
	.MLIST	p	Control macro listing
	.XLIST	p	Control macro object listing
	.ERROR	s	List an error message
	.WARN	s	List a warning message
	.PAGE	n?	Start a new page
	.LIBRARY	p l? ...	Library search
	.OCOMMENT	n? s	Object comment
Macro Operations			
ll	.MACRO	f ...	Define macro
	.ENDM		End MACRO definition
ll	.PROC	l ...	Define a procedure
	.ENDP		End PROC definition
	.EXITM	n?	End macro expansion
	.LOCAL	l ...	Define macro labels
	.REPT	n	Repeat
	.IRP	f s	Repeat on parameter list
	.IRPC	f s	Repeat on character string
	.ENDM		End repeated block

APPENDIX B
SPECIAL SYMBOLS

The following special symbols are defined. They can be reassigned using .SET pseudo-ops, and can be used as operands anywhere a label could be used. If needed, additional special symbols will be defined later.

Symbol	Initial Value	Meaning
$'LIST	1	Controls the whole listing
$'WLIST	1	Controls the warning listing
$'CLIST	1	Controls listing of false conditional
$'MLIST	1	Controls macro expansion listing
$'XLIST	1	Controls listing of object code that does not fit on original source line

These special symbols are used for control of the listing. If the low-order bit is 1, the corresponding item is listed. If the low-order bit is 0, the item is not listed.

$'LIST controls the listing as a whole, $'WLIST controls the listing of warning messages, $'CLIST the listing of false conditionals, $'MLIST the listing of macro expansions, and $'XLIST the listing of object code that does not fit on the original source line.

	Default Value
$'SYMLEN	127

The maximum number of significant characters in a symbol.

$'UCASE 0

Treat all letters as uppercase.

$'STRESC '\'

The string-escape character. The meaning of the following character is given in the table in section 3.3.2 (constants).

$'S1LEN	10
$'S1ORD	'M'
$'S2LEN	10
$'S2ORD	'M'

The length and byte-order ('M' = most significant byte first, 'L' = least significant byte first) of strings surrounded by single and double quotes respectively. In the byte-order parameters, only the least-significant bit is actually looked at. Thus, 0 and 1 can be used instead of 'L' and 'M', respectively.

$'SxLEN and $'SxORD are provided because previous Z8000 assemblers have evaluated byte order differently when using strings as numbers.

$'BASE	10
$'ZBASE	10
$'SBASE	8

The input default number base for numbers that start with non-zero digits, numbers that start with zero, and string escape sequences respectively. Setting $'ZBASE to 8 gives the C convention for octal numbers. Terms like $'BASE must be in the range 2 to 16.

$'ADRLEN 2

The length in bytes of an address. The value for $'ADRLEN is 2.

$'ADRORD 'M'

The byte-order of an address. $'ADRORD is normally left as 'M'; this can be changed if the assembler is being used to produce non-Z80,000 code.

$'ADRTYPE 0

This indicates the current addressing type: 0 = linear, 1 = segmented, 2 = compact (nonsegmented).

$'ALIGN 1

The alignment boundary for instructions and data with length >= 1 byte.

$'EPUID 0

The current EPU Identifier. Unused.

$'Z8 0 (1 if -r option)

When set to 1, the Super8 instruction set is accepted. When cleared to 0 (explicitly or with an option), the Z8 instruction set is accepted.

$'OPCOPT 0

If the value is not zero and an opcode is missing on a line containing expressions, the opcode .DD (arbitrary-length data) will be assumed.

APPENDIX C
ASCII CHARACTER SET

	Numeric		
Graphic	Decimal	Hex	Comments
	0	0	Null
	1	1	Start of heading
	2	2	Start of text
	3	3	End of text
	4	4	End of transmission
	5	5	Enquiry
	6	6	Acknowledge
	7	7	Bell
	8	8	Backspace
	9	9	Horizontal tabulation
	10	A	Line feed
	11	B	Vertical tabulation
	12	C	Form feed
	13	D	Carriage return
	14	E	Shift out
	15	F	Shift in
	16	10	Data link escape
	17	11	Device control 1
	18	12	Device control 2
	19	13	Device control 3
	20	14	Device control 4
	21	15	Negative acknowledge
	22	16	Synchronous idle
	23	17	End of block
	24	18	Cancel
	25	19	End of medium
	26	1A	Substitute
	27	1B	Escape
	28	1C	File separator
	29	1D	Group separator
	30	1E	Record separator
	31	1F	Unit separator
	32	20	Space
!	33	21	Exclamation point
"	34	22	Quotation mark
#	35	23	Number sign
$	36	24	Dollar sign
%	37	25	Percent sign
&	38	26	Ampersand
'	39	27	Apostrophe
(	40	28	Opening parenthesis
)	41	29	Closing parenthesis
*	42	2A	Asterisk
+	43	2B	Plus
,	44	2C	Comma

ASCII Character Set
(Continued)

	Numeric		
Graphic	**Decimal**	**Hex**	**Comments**
-	45	2D	Hyphen (minus)
.	46	2E	Period (decimal point)
/	47	2F	Slant
0	48	30	Zero
1	49	31	One
2	50	32	Two
3	51	33	Three
4	52	34	Four
5	53	35	Five
6	54	36	Six
7	55	37	Seven
8	56	38	Eight
9	57	39	Nine
:	58	3A	Colon
;	59	3B	Semicolon
<	60	3C	Less than
=	61	3D	Equals
>	62	3E	Greater than
?	63	3F	Question mark
@	64	40	Commercial at
A	65	41	Uppercase A
B	66	42	Uppercase B
C	67	43	Uppercase C
D	68	44	Uppercase D
E	69	45	Uppercase E
F	70	46	Uppercase F
G	71	47	Uppercase G
H	72	48	Uppercase H
I	73	49	Uppercase I
J	74	4A	Uppercase J
K	75	4B	Uppercase K
L	76	4C	Uppercase L
M	77	4D	Uppercase M
N	78	4E	Uppercase N
O	79	4F	Uppercase O
P	80	50	Uppercase P
Q	81	51	Uppercase Q
R	82	52	Uppercase R
S	83	53	Uppercase S
T	84	54	Uppercase T
U	85	55	Uppercase U
V	86	56	Uppercase V
W	87	57	Uppercase W
X	88	58	Uppercase X
Y	89	59	Uppercase Y
Z	90	5A	Uppercase Z
[	91	5B	Opening bracket
\	92	5C	Reverse slant
]	93	5D	Closing bracket
^	94	5E	Circumflex
_	95	5F	Underscore
`	96	60	Grave accent

ASCII Character Set (Continued)

Graphic	Numeric Decimal	 Hex	Comments
a	97	61	Lowercase a
b	98	62	Lowercase b
c	99	63	Lowercase c
d	100	64	Lowercase d
e	101	65	Lowercase e
f	102	66	Lowercase f
g	103	67	Lowercase g
h	104	68	Lowercase h
i	105	69	Lowercase i
j	106	6A	Lowercase j
k	107	6B	Lowercase k
l	108	6C	Lowercase l
m	109	6D	Lowercase m
n	110	6E	Lowercase n
o	111	6F	Lowercase o
p	112	70	Lowercase p
q	113	71	Lowercase q
r	114	72	Lowercase r
s	115	73	Lowercase s
t	116	74	Lowercase t
u	117	75	Lowercase u
v	118	76	Lowercase v
w	119	77	Lowercase w
x	120	78	Lowercase x
y	121	79	Lowercase y
z	122	7A	Lowercase z
{	123	7B	Opening (left) brace
|	124	7C	Vertical line
}	125	7D	Closing (right) brace
~	126	7E	Tilde
	127	7F	Delete

APPENDIX D
ERROR MESSAGES AND EXPLANATIONS

ENDIF (end conditional) expected

.IF was seen but not followed by a matching .ENDIF.

ENDM (end macro definition) expected

End of file was reached while still inside a macro definition.

can't set read-only symbol

An attempt was made to set a special symbol such as $'PASS, that cannot be redefined.

extended instruction set not allowed

An attempt was made to use a Super8 instruction or addressing mode not available on the Z8 CPU while the -r option or $' Z8 flag is in effect.

extra parameters (ignored)

A pseudo-op was passed more parameters than it requires. The extra parameters will be ignored.

extra right parenthesis (ignored)

A right parenthesis was seen without a matching left parenthesis. It is ignored.

forward reference not allowed here

An expression in an IF, COND, EQU, or SET contains a forward reference (a label that has not been defined earlier in the program).

label required

A pseudo-op such as EQU or SET, which require a label, does not have one.

line too long (truncated)

The source file or a macro expansion contains a line longer than 512 characters.

link-time expression not allowed here

An expression that cannot be evaluated by the assembler has been used in a context where the assembler needs to know its value.

missing parameter

A pseudo-op has been given fewer parameters than it requires.

missing right parenthesis (assumed)

The end of an expression was encountered without finding a right parenthesis to match a left parenthesis already seen. The assembler will evaluate the expression as if the missing parenthesis had been at the end of the expression.

Multiple definition

A symbol has been used as a label, defined by an EQU, or defined as a macro more than once.

no input file

The assembler cannot open the specified input file.

operand expected (0 assumed)

A binary expression operator (such as +) was not followed by an operand. A zero operand is assumed.

operation not defined on register

An expression operator (such as *) has been applied to a register value for which it is not valid. The only expression operators that can be applied to registers are indexing and indirection.

parser stack overflow

The assembler received an expression too complex for it to handle.

phase error--passes out of sync.

Something happened differently on passes 1 and 2 of the assembler. This can occur if an opcode or pseudo-op is used and later redefined as a macro.

storage allocation failed

The assembler ran out of storage as a result of a combination of symbol table, macro definitions, and macro invocations.

syntax error

A source statement contains a syntactic error, usually in an expression, which cannot be otherwise classified.

undefined addressing mode expression

An expression represents an addressing mode not available on the Super8 and Z8 CPU, such as (HL + A).

undefined character

A character appears in the input that the assembler does not understand.

undefined symbol

A symbol has been used that is never defined. The value 0 will normally be used.

value out of range

An expression does not fit in the specified size of field (for example, an address in a .BYTE statement).

wrong operand type for this operation

An opcode has been given an operand with an addressing mode that does not apply to it.

```
asmS8 version 1.0           t.z8inst

LOC       OBJ             LINE# --- SOURCE ---
00000000 1235                 1         adc     r3,r5
00000002 1335                 2         adc     r3,@r5
00000004 1440e3               3         adc     r3,64
00000007 14e520               4         adc     32,r5
0000000a 144020               5         adc     32,64
0000000d 1540e3               6         adc     r3,@64
00000010 15e520               7         adc     32,@r5
00000013 154020               8         adc     32,@64
00000016 16e340               9         adc     r3,#64
00000019 162040              10         adc     32,#64
0000001c 172040              11         adc     @32,#64
0000001f 17e340              12         adc     @r3,#64
                             13
00000022 0235                14         add     r3,r5
00000024 0335                15         add     r3,@r5
00000026 0440e3              16         add     r3,64
00000029 04e520              17         add     32,r5
0000002c 044020              18         add     32,64
0000002f 0540e3              19         add     r3,@64
00000032 05e520              20         add     32,@r5
00000035 054020              21         add     32,@64
00000038 06e340              22         add     r3,#64
0000003b 062040              23         add     32,#64
0000003e 072040              24         add     @32,#64
00000041 07e340              25         add     @r3,#64
                             26
00000044 5235                27         and     r3,r5
00000046 5335                28         and     r3,@r5
00000048 5440e3              29         and     r3,64
0000004b 54e520              30         and     32,r5
0000004e 544020              31         and     32,64
00000051 5540e3              32         and     r3,@64
00000054 55e520              33         and     32,@r5
00000057 554020              34         and     32,@64
0000005a 56e340              35         and     r3,#64
0000005d 562040              36         and     32,#64
00000060 572040              37         and     @32,#64
00000063 57e340              38         and     @r3,#64
                             39
00000066 d4e2                40         call    @rr2
00000068 d420                41         call    @32
0000006a d60040              42         call    64
                             43
0000006d ef                  44         ccf
                             45
0000006e b0e3                46         clr     r3
00000070 b020                47         clr     32
00000072 b1e3                48         clr     @r3
00000074 b120                49         clr     @32
                             50
00000076 60e3                51         com     r3
00000078 6020                52         com     32
```

```
0000007a 61e3           53        com     @r3
0000007c 6120           54        com     @32
                        55
0000007e a235           56        cp      r3,r5
00000080 a335           57        cp      r3,@r5
00000082 a440e3         58        cp      r3,64
00000085 a4e520         59        cp      32,r5
00000088 a44020         60        cp      32,64
0000008b a540e3         61        cp      r3,@64
0000008e a5e520         62        cp      32,@r5
00000091 a54020         63        cp      32,@64
00000094 a6e340         64        cp      r3,#64
00000097 a72040         65        cp      @32,#64
0000009a a7e340         66        cp      @r3,#64
                        67
0000009d 40e3           68        da      r3
0000009f 4020           69        da      32
000000a1 41e3           70        da      @r3
000000a3 4120           71        da      @32
                        72
000000a5 00e3           73        dec     r3
000000a7 0020           74        dec     32
000000a9 01e3           75        dec     @r3
000000ab 0120           76        dec     @32
                        77
000000ad 80e2           78        decw    rr2
000000af 8020           79        decw    32
000000b1 81e3           80        decw    @r3
000000b3 8120           81        decw    @32
                        82
000000b5 8f             83        di
                        84
000000b6 3afe           85        djnz    r3,$
                        86
000000b8 9f             87        ei
                        88
000000b9 3e             89        inc     r3
000000ba 2020           90        inc     32
000000bc 21e3           91        inc     @r3
000000be 2120           92        inc     @32
                        93
000000c0 a0e2           94        incw    rr2
000000c2 a020           95        incw    32
000000c4 a1e3           96        incw    @r3
000000c6 a120           97        incw    @32
                        98
000000c8 bf             99        iret
                       100
000000c9 8d0400        101        jp      1024
000000cc ed0400        102        jp      nz,1024
000000cf 30e2          103        jp      @rr2
000000d1 3020          104        jp      @32
                       105
000000d3 8bfe          106        jr      $
000000d5 ebfe          107        jr      nz,$
```

```
                                108
000000d7 3c40                   109          ld       r3,#64
                                110
000000d9 38e5                   111          ld       r3,r5
000000db 3840                   112          ld       r3,64
000000dd 5920                   113          ld       32,r5
                                114
000000df e335                   115          ld       r3,@r5
000000e1 f335                   116          ld       @r3,r5
                                117
000000e3 e44020                 118          ld       32,64
                                119
000000e6 e335                   120          ld       r3,@r5
000000e8 e540e3                 121          ld       r3,@64
000000eb e5e520                 122          ld       32,@r5
000000ee e54020                 123          ld       32,@64
                                124
000000f1 3c40                   125          ld       r3,#64
000000f3 e62040                 126          ld       32,#64
000000f6 e7e340                 127          ld       @r3,#64
000000f9 d62040                 128          ld       @32,#64
                                129
000000fc f335                   130          ld       @r3,r5
000000fe f540e3                 131          ld       @r3,64
00000101 f5e520                 132          ld       @32,r5
00000104 f54020                 133          ld       @32,64
                                134
00000107 c73540                 135          ld       r3,64(r5)
0000010a d75340                 136          ld       64(r3),r5
                                137
0000010d c234                   138          ldc      r3,@rr4
0000010f d252                   139          ldc      @rr2,r5
                                140
00000111 c334                   141          ldci     @r3,@rr4
00000113 d352                   142          ldci     @rr2,@r5
                                143
00000115 8234                   144          lde      r3,@rr4
00000117 9252                   145          lde      @rr2,r5
                                146
00000119 9352                   147          ldei     @rr2,@r5
0000011b 8334                   148          ldei     @r3,@rr4
                                149
0000011d ff                     150          nop
                                151
0000011e 4235                   152          or       r3,r5
00000120 4335                   153          or       r3,@r5
00000122 4440e3                 154          or       r3,64
00000125 44e520                 155          or       32,r5
00000128 444020                 156          or       32,64
0000012b 4540e3                 157          or       r3,@64
0000012e 45e520                 158          or       32,@r5
00000131 454020                 159          or       32,@64
00000134 46e340                 160          or       r3,#64
00000137 462040                 161          or       32,#64
0000013a 472040                 162          or       @32,#64
```

```
0000013d 47e340          163       or     @r3,#64
                         164
00000140 50e3            165       pop    r3
00000142 5020            166       pop    32
00000144 51e3            167       pop    @r3
00000146 5120            168       pop    @32
                         169
00000148 70e3            170       push   r3
0000014a 7020            171       push   32
0000014c 71e3            172       push   @r3
0000014e 7120            173       push   @32
                         174
00000150 cf              175       rcf
                         176
00000151 af              177       ret
                         178
00000152 90e3            179       rl     r3
00000154 9020            180       rl     32
00000156 91e3            181       rl     @r3
00000158 9120            182       rl     @32
                         183
0000015a 10e3            184       rlc    r3
0000015c 1020            185       rlc    32
0000015e 11e3            186       rlc    @r3
00000160 1120            187       rlc    @32
                         188
00000162 e0e3            189       rr     r3
00000164 e020            190       rr     32
00000166 ele3            191       rr     @r3
00000168 el20            192       rr     @32
                         193
0000016a c0e3            194       rrc    r3
0000016c c020            195       rrc    32
0000016e cle3            196       rrc    @r3
00000170 cl20            197       rrc    @32
                         198
00000172 3235            199       sbc    r3,r5
00000174 3335            200       sbc    r3,@r5
00000176 3440e3          201       sbc    r3,64
00000179 34e520          202       sbc    32,r5
0000017c 344020          203       sbc    32,64
0000017f 3540e3          204       sbc    r3,@64
00000182 35e520          205       sbc    32,@r5
00000185 354020          206       sbc    32,@64
00000188 36e340          207       sbc    r3,#64
0000018b 362040          208       sbc    32,#64
0000018e 372040          209       sbc    @32,#64
00000191 37e340          210       sbc    @r3,#64
                         211
00000194 df              212       scf
                         213
00000195 d0e3            214       sra    r3
00000197 d020            215       sra    32
00000199 dle3            216       sra    @r3
0000019b dl20            217       sra    @32
```

```
                                218
0000019d 3170                   219          srp      #70h
                                220
0000019f 2235                   221          sub      r3,r5
000001a1 2335                   222          sub      r3,@r5
000001a3 2440e3                 223          sub      r3,64
000001a6 24e520                 224          sub      32,r5
000001a9 244020                 225          sub      32,64
000001ac 2540e3                 226          sub      r3,@64
000001af 25e520                 227          sub      32,@r5
000001b2 254020                 228          sub      32,@64
000001b5 26e340                 229          sub      r3,#64
000001b8 262040                 230          sub      32,#64
000001bb 272040                 231          sub      @32,#64
000001be 27e340                 232          sub      @r3,#64
                                233
000001c1 f0e3                   234          swap     r3
000001c3 f020                   235          swap     32
000001c5 f1e3                   236          swap     @r3
000001c7 f120                   237          swap     @32
                                238
000001c9 6235                   239          tcm      r3,r5
000001cb 6335                   240          tcm      r3,@r5
000001cd 6440e3                 241          tcm      r3,64
000001d0 64e520                 242          tcm      32,r5
000001d3 644020                 243          tcm      32,64
000001d6 6540e3                 244          tcm      r3,@64
000001d9 65e520                 245          tcm      32,@r5
000001dc 654020                 246          tcm      32,@64
000001df 66e340                 247          tcm      r3,#64
000001e2 662040                 248          tcm      32,#64
000001e5 672040                 249          tcm      @32,#64
000001e8 67e340                 250          tcm      @r3,#64
                                251
000001eb 7235                   252          tm       r3,r5
000001ed 7335                   253          tm       r3,@r5
                                254
000001ef 7440e3                 255          tm       r3,64
000001f2 74e520                 256          tm       32,r5
000001f5 744020                 257          tm       32,64
000001f8 7540e3                 258          tm       r3,@64
000001fb 75e520                 259          tm       32,@r5
000001fe 754020                 260          tm       32,@64
00000201 76e340                 261          tm       r3,#64
00000204 762040                 262          tm       32,#64
00000207 772040                 263          tm       @32,#64
0000020a 77e340                 264          tm       @r3,#64
                                265
0000020d b235                   266          xor      r3,r5
0000020f b335                   267          xor      r3,@r5
00000211 b440e3                 268          xor      r3,64
00000214 b4e520                 269          xor      32,r5
00000217 b44020                 270          xor      32,64
0000021a b540e3                 271          xor      r3,@64
0000021d b5e520                 272          xor      32,@r5
```

```
00000220 b54020          273          xor       32,@64
00000223 b6c340          274          xor       r3,#64
00000226 b62040          275          xor       32,#64
00000229 b72040          276          xor       @32,#64
0000022c b7e340          277          xor       @r3,#64
                         278
                         279
                         280 ;defined register names
                         281
0000022f 38ff            282          ld        r3,spl
00000231 38fe            283          ld        r3,sph
00000233 38fd            284          ld        r3,rp
00000235 38fc            285          ld        r3,flags
00000237 38fb            286          ld        r3,imr
00000239 38fa            287          ld        r3,irq
0000023b 38f9            288          ld        r3,ipr
0000023d 38f8            289          ld        r3,p01m
0000023f 38f7            290          ld        r3,p3m
00000241 38f6            291          ld        r3,p2m
00000243 38f5            292          ld        r3,pre0
00000245 38f4            293          ld        r3,t0
00000247 38f3            294          ld        r3,pre1
00000249 38f2            295          ld        r3,t1
0000024b 38f1            296          ld        r3,tmr
0000024d 38f0            297          ld        r3,sio
0000024f 3803            298          ld        r3,p3
00000251 3802            299          ld        r3,p2
00000253 3801            300          ld        r3,p1
00000255 3800            301          ld        r3,p0
                         302
                         303
                         304 ;defined register names
                         305
00000257 38ff            306          ld        r3,SPL
00000259 38fe            307          ld        r3,SPH
0000025b 38fd            308          ld        r3,RP
0000025d 38fc            309          ld        r3,FLAGS
0000025f 38fb            310          ld        r3,IMR
00000261 38fa            311          ld        r3,IRQ
00000263 38f9            312          ld        r3,IPR
00000265 38f8            313          ld        r3,P01M
00000267 38f7            314          ld        r3,P3M
00000269 38f6            315          ld        r3,P2M
0000026b 38f5            316          ld        r3,PRE0
0000026d 38f4            317          ld        r3,T0
0000026f 38f3            318          ld        r3,PRE1
00000271 38f2            319          ld        r3,T1
00000273 38f1            320          ld        r3,TMR
00000275 38f0            321          ld        r3,SIO
00000277 3803            322          ld        r3,P3
00000279 3802            323          ld        r3,P2
0000027b 3801            324          ld        r3,P1
0000027d 3800            325          ld        r3,P0
                         326
                         327 ;test for condition codes
```

```
                                328
0000027f 0d0080                 329           jp        f,128
                                330
00000282 6d0080                 331           jp        z,128
00000285 ed0080                 332           jp        nz,128
00000288 6d0080                 333           jp        eq,128
0000028b ed0080                 334           jp        ne,128
                                335
0000028e 7d0080                 336           jp        c,128
00000291 fd0080                 337           jp        nc,128
                                338
00000294 ad0080                 339           jp        gt,128
00000297 1d0080                 340           jp        lt,128
0000029a 9d0080                 341           jp        ge,128
0000029d 2d0080                 342           jp        le,128
                                343
000002a0 dd0080                 344           jp        pl,128
000002a3 5d0080                 345           jp        mi,128
                                346
000002a6 cd0080                 347           jp        nov,128
000002a9 4d0080                 348           jp        ov,128
                                349
000002ac bd0080                 350           jp        ugt,128
000002af 7d0080                 351           jp        ult,128
000002b2 fd0080                 352           jp        uge,128
000002b5 3d0080                 353           jp        ule,128
                                354
```

```
asmS8 version 1.0
                        t.s8inst
LOC      OBJ       LINE# --- SOURCE ---
                       1 ;reference test source for Super8 instructin set.
                       2
                       3
00000000 1235          4         adc     r3,r5
00000002 1335          5         adc     r3,@r5
00000004 1440c3        6         adc     r3,64
00000007 14c520        7         adc     32,r5
0000000a 144020        8         adc     32,64
0000000d 1540c3        9         adc     r3,@64
00000010 15c520       10         adc     32,@r5
00000013 154020       11         adc     32,@64
00000016 16c340       12         adc     r3,#64
00000019 162040       13         adc     32,#64
                      14
0000001c 0235         15         add     r3,r5
0000001e 0335         16         add     r3,@r5
00000020 0440c3       17         add     r3,64
00000023 04c520       18         add     32,r5
00000026 044020       19         add     32,64
00000029 0540c3       20         add     r3,@64
0000002c 05c520       21         add     32,@r5
0000002f 054020       22         add     32,@64
00000032 06c340       23         add     r3,#64
00000035 062040       24         add     32,#64
                      25
00000038 5235         26         and     r3,r5
0000003a 5335         27         and     r3,@r5
0000003c 5440c3       28         and     r3,64
0000003f 54c520       29         and     32,r5
00000042 544020       30         and     32,64
00000045 5540c3       31         and     r3,@64
00000048 55c520       32         and     32,@r5
0000004b 554020       33         and     32,@64
0000004e 56c340       34         and     r3,#64
00000051 562040       35         and     32,#64
                      36
00000054 673ec5       37         band    r3,r5,#7
00000057 673e40       38         band    r3,64,#7
0000005a 675fc3       39         band    r3,#7,r5
0000005d 675f20       40         band    32,#7,r5
                      41
00000060 173ec5       42         bcp     r3,r5,#7
00000063 173e40       43         bcp     r3,64,#7
                      44
00000066 573e         45         bitc    r3,#7
                      46
00000068 773e         47         bitr    r3,#7
                      48
0000006a 773f         49         bits    r3,#7
                      50
0000006c 073ec5       51         bor     r3,r5,#7
0000006f 073e40       52         bor     r3,64,#7
```

```
00000072 075fc3          53        bor    r3,#7,r5
00000075 075f20          54        bor    32,#7,r5
                         55
00000078 375efd          56        btjrf  $,r5,#7
0000007b 375ffd          57        btjrt  $,r5,#7
                         58
0000007e 273ec5          59        bxor   r3,r5,#7
00000081 273e40          60        bxor   r3,64,#7
00000084 275fc3          61        bxor   r3,#7,r5
00000087 275f20          62        bxor   32,#7,r5
                         63
0000008a d420            64        call   #32
0000008c f4c2            65        call   @rr2
0000008e f420            66        call   @32
00000090 f60040          67        call   64
                         68
00000093 ef              69        ccf
                         70
00000094 b0c3            71        clr    r3
00000096 b020            72        clr    32
00000098 b1c3            73        clr    @r3
0000009a b120            74        clr    @32
                         75
0000009c 60c3            76        com    r3
0000009e 6020            77        com    32
000000a0 61c3            78        com    @r3
000000a2 6120            79        com    @32
                         80
000000a4 a235            81        cp     r3,r5
000000a6 a335            82        cp     r3,@r5
000000a8 a440c3          83        cp     r3,64
000000ab a4c520          84        cp     32,r5
000000ae a44020          85        cp     32,64
000000b1 a540c3          86        cp     r3,@64
000000b4 a5c520          87        cp     32,@r5
000000b7 a54020          88        cp     32,@64
000000ba a6c340          89        cp     r3,#64
                         90
000000bd d253fd          91        cpijne r3,@r5,$
                         92
000000c0 c253fd          93        cpije  r3,@r5,$
                         94
000000c3 40c3            95        da     r3
000000c5 4020            96        da     32
000000c7 41c3            97        da     @r3
000000c9 4120            98        da     @32
                         99
000000cb 00c3           100        dec    r3
000000cd 0020           101        dec    32
000000cf 01c3           102        dec    @r3
000000d1 0120           103        dec    @32
                        104
000000d3 80c2           105        decw   rr2
000000d5 8020           106        decw   32
000000d7 81c3           107        decw   @r3
```

```
000000d9 8120          decw   @32

000000db 8f            di

000000dc 94c5c2        div    rr2,r5
000000df 9440c2        div    rr2,64
000000e2 94c520        div    32,r5
000000e5 944020        div    32,64
000000e8 95c5c2        div    rr2,@r5
000000eb 9540c2        div    rr2,@64
000000ee 95c520        div    32,@r5
000000f1 954020        div    32,@64
000000f4 9640c2        div    rr2,#64
000000f7 964020        div    32,#64

000000fa 3afe          djnz   r3,$

000000fc 9f            ei

000000fd 1f            enter

000000fe 2f            exit

000000ff 3e            inc    r3
00000100 2020          inc    32
00000102 21c3          inc    @r3
00000104 2120          inc    @32

00000106 a0c2          incw   rr2
00000108 a020          incw   32
0000010a a1c3          incw   @r3
0000010c a120          incw   @32

0000010e bf            iret

0000010f 8d0400        jp     1024
00000112 ed0400        jp     nz,1024
00000115 30c2          jp     @rr2
00000117 3020          jp     @32

00000119 8bfe          jr     $
0000011b ebfe          jr     nz,$

0000011d 3c40          ld     r3,#64

0000011f 38c5          ld     r3,r5
00000121 3840          ld     r3,64
00000123 5920          ld     32,r5

00000125 c735          ld     r3,@r5
00000127 d735          ld     @r3,r5

00000129 e44020        ld     32,64

0000012c c735          ld     r3,@r5
```

```
0000012e e540c3         163        ld     r3,@64
00000131 e5c520         164        ld     32,@r5
00000134 e54020         165        ld     32,@64
                        166
00000137 3c40           167        ld     r3,#64
00000139 e62040         168        ld     32,#64
0000013c d6c340         169        ld     @r3,#64
0000013f d62040         170        ld     @32,#64
                        171
00000142 d735           172        ld     @r3,r5
00000144 f540c3         173        ld     @r3,64
00000147 f5c520         174        ld     @32,r5
0000014a f54020         175        ld     @32,64
                        176
0000014d 873540         177        ld     r3,64(r5)
00000150 975340         178        ld     64(r3),r5
                        179
00000153 473ec5         180        ldb    r3,r5,#7
00000156 473e40         181        ldb    r3,64,#7
00000159 475fc3         182        ldb    r3,#7,r5
0000015c 475f20         183        ldb    32,#7,r5
                        184
0000015f a7340004       185        ldc    r3,1024(rr4)
00000163 e73440         186        ldc    r3,64(rr4)
00000166 b7520004       187        ldc    1024(rr2),r5
0000016a f75240         188        ldc    64(rr2),r5
0000016d b7500020       189        ldc    32,r5
00000171 a7500040       190        ldc    r5,64
00000175 c334           191        ldc    r3,@rr4
00000177 d352           192        ldc    @rr2,r5
                        193
00000179 e234           194        ldcd   r3,@rr4
0000017b e334           195        ldci   r3,@rr4
0000017d f252           196        ldcpd  @rr2,r5
0000017f f352           197        ldcpi  @rr2,r5
                        198
00000181 a7350004       199        lde    r3,1024(rr4)
00000185 e73540         200        lde    r3,64(rr4)
00000188 b7530004       201        lde    1024(rr2),r5
0000018c f75340         202        lde    64(rr2),r5
0000018f b7510020       203        lde    32,r5
00000193 a7510040       204        lde    r5,64
00000197 c335           205        lde    r3,@rr4
00000199 d353           206        lde    @rr2,r5
                        207
0000019b e235           208        lded   r3,@rr4
0000019d e335           209        ldei   r3,@rr4
0000019f f253           210        ldepd  @rr2,r5
000001a1 f353           211        ldepi  @rr2,r5
                        212
000001a3 c4c4c2         213        ldw    rr2,rr4
000001a6 c440c2         214        ldw    rr2,64
000001a9 c4c420         215        ldw    32,rr4
000001ac c44020         216        ldw    32,64
                        217
```

```
000001af c5c4c2          218          ldw     rr2,@r4
000001b2 c540c2          219          ldw     rr2,@64
000001b5 c5c420          220          ldw     32,@r4
000001b8 c54020          221          ldw     32,@64
                         222
000001bb c6c20400        223          ldw     rr2,#1024
000001bf c6200400        224          ldw     32,#1024
                         225
000001c3 84c5c2          226          mult    rr2,r5
000001c6 8440c2          227          mult    rr2,64
000001c9 84c520          228          mult    32,r5
000001cc 844020          229          mult    32,64
000001cf 85c5c2          230          mult    rr2,@r5
000001d2 8540c2          231          mult    rr2,@64
000001d5 85c520          232          mult    32,@r5
000001d8 854020          233          mult    32,@64
000001db 8640c2          234          mult    rr2,#64
000001de 864020          235          mult    32,#64
                         236
000001e1 0f              237          next
                         238
000001e2 ff              239          nop
                         240
000001e3 4235            241          or      r3,r5
000001e5 4335            242          or      r3,@r5
000001e7 4440c3          243          or      r3,64
000001ea 44c520          244          or      32,r5
000001ed 444020          245          or      32,64
000001f0 4540c3          246          or      r3,@64
000001f3 45c520          247          or      32,@r5
000001f6 454020          248          or      32,@64
000001f9 46c340          249          or      r3,#64
000001fc 462040          250          or      32,#64
                         251
000001ff 50c3            252          pop     r3
00000201 5020            253          pop     32
00000203 51c3            254          pop     @r3
00000205 5120            255          pop     @32
                         256
00000207 92c5c3          257          popud   r3,@r5
0000020a 9240c3          258          popud   r3,@64
0000020d 92c520          259          popud   32,@r5
00000210 924020          260          popud   32,@64
                         261
00000213 93c5c3          262          popui   r3,@r5
00000216 9340c3          263          popui   r3,@64
00000219 93c520          264          popui   32,@r5
0000021c 934020          265          popui   32,@64
                         266
0000021f 70c3            267          push    r3
00000221 7020            268          push    32
00000223 71c3            269          push    @r3
00000225 7120            270          push    @32
                         271
00000227 82c3c5          272          pushud  @r3,r5
```

```
0000022a 82c340          273          pushud @r3,64
0000022d 8220c5          274          pushud @32,r5
00000230 822040          275          pushud @32,64
                         276
00000233 83c3c5          277          pushui @r3,r5
00000236 83c340          278          pushui @r3,64
00000239 8320c5          279          pushui @32,r5
0000023c 832040          280          pushui @32,64
                         281
0000023f cf              282          rcf
                         283
00000240 d5a5            284          rdr    #0a5h
                         285
00000242 af              286          ret
                         287
00000243 90c3            288          rl     r3
00000245 9020            289          rl     32
00000247 91c3            290          rl     @r3
00000249 9120            291          rl     @32
                         292
0000024b 10c3            293          rlc    r3
0000024d 1020            294          rlc    32
0000024f 11c3            295          rlc    @r3
00000251 1120            296          rlc    @32
                         297
00000253 e0c3            298          rr     r3
00000255 e020            299          rr     32
00000257 e1c3            300          rr     @r3
00000259 e120            301          rr     @32
                         302
0000025b c0c3            303          rrc    r3
0000025d c020            304          rrc    32
0000025f c1c3            305          rrc    @r3
00000261 c120            306          rrc    @32
                         307
00000263 4f              308          sb0
                         309
00000264 5f              310          sb1
                         311
00000265 3235            312          sbc    r3,r5
00000267 3335            313          sbc    r3,@r5
00000269 3440c3          314          sbc    r3,64
0000026c 34c520          315          sbc    32,r5
0000026f 344020          316          sbc    32,64
00000272 3540c3          317          sbc    r3,@64
00000275 35c520          318          sbc    32,@r5
00000278 354020          319          sbc    32,@64
0000027b 36c340          320          sbc    r3,#64
0000027e 362040          321          sbc    32,#64
                         322
00000281 df              323          scf
                         324
00000282 d0c3            325          sra    r3
00000284 d020            326          sra    32
00000286 d1c3            327          sra    @r3
```

```
00000288 d120          328        sra     @32
                       329
0000028a 3180          330        srp     #128
0000028c 3181          331        srp1    #128
0000028e 3182          332        srp0    #128
                       333
00000290 2235          334        sub     r3,r5
00000292 2335          335        sub     r3,@r5
00000294 2440c3        336        sub     r3,64
00000297 24c520        337        sub     32,r5
0000029a 244020        338        sub     32,64
0000029d 2540c3        339        sub     r3,@64
000002a0 25c520        340        sub     32,@r5
000002a3 254020        341        sub     32,@64
000002a6 26c340        342        sub     r3,#64
000002a9 262040        343        sub     32,#64
                       344
000002ac f0c3          345        swap    r3
000002ae f020          346        swap    32
000002b0 f1c3          347        swap    @r3
000002b2 f120          348        swap    @32
                       349
000002b4 6235          350        tcm     r3,r5
000002b6 6335          351        tcm     r3,@r5
000002b8 6440c3        352        tcm     r3,64
000002bb 64c520        353        tcm     32,r5
000002be 644020        354        tcm     32,64
000002c1 6540c3        355        tcm     r3,@64
000002c4 65c520        356        tcm     32,@r5
000002c7 654020        357        tcm     32,@64
000002ca 66c340        358        tcm     r3,#64
000002cd 662040        359        tcm     32,#64
                       360
000002d0 7235          361        tm      r3,r5
000002d2 7335          362        tm      r3,@r5
                       363
000002d4 7440c3        364        tm      r3,64
000002d7 74c520        365        tm      32,r5
000002da 744020        366        tm      32,64
000002dd 7540c3        367        tm      r3,@64
000002e0 75c520        368        tm      32,@r5
000002e3 754020        369        tm      32,@64
000002e6 76c340        370        tm      r3,#64
000002e9 762040        371        tm      32,#64
                       372
000002ec b235          373        xor     r3,r5
000002ee b335          374        xor     r3,@r5
000002f0 b440c3        375        xor     r3,64
000002f3 b4c520        376        xor     32,r5
000002f6 b44020        377        xor     32,64
000002f9 b540c3        378        xor     r3,@64
000002fc b5c520        379        xor     32,@r5
000002ff b54020        380        xor     32,@64
00000302 b6c340        381        xor     r3,#64
00000305 b62040        382        xor     32,#64
```

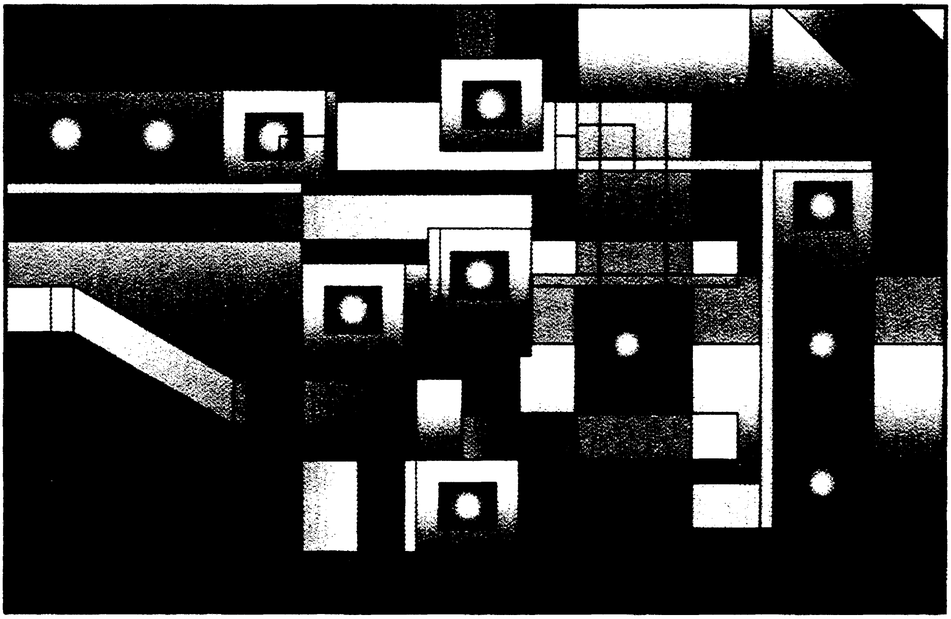
```
                           383
00000308 3f                384           wfi
                           385
                           386 ;defined register names
                           387
00000309 38de              388          ld       r3,sym
0000030b 38dd              389          ld       r3,imr
0000030d 38dc              390          ld       r3,irr
0000030f c4dac2            391          ldw      rr2,ip
00000312 38db              392          ld       r3,ipl
00000314 38da              393          ld       r3,iph
00000316 c4d8c2            394          ldw      rr2,sp
00000319 38d9              395          ld       r3,spl
0000031b 38d8              396          ld       r3,sph
0000031d 38d7              397          ld       r3,rp1
0000031f 38d6              398          ld       r3,rp0
00000321 38d5              399          ld       r3,flags
00000323 38d4              400          ld       r3,p4
00000325 38d3              401          ld       r3,p3
00000327 38d2              402          ld       r3,p2
00000329 38d1              403          ld       r3,p1
0000032b 38d0              404          ld       r3,p0
                           405
                           406 ; Bank 0 Special Registers
                           407
0000032d 38ff              408          ld       r3,ipr
0000032f 38fe              409          ld       r3,emt
00000331 38fd              410          ld       r3,p2bip
00000333 38fc              411          ld       r3,p2aip
00000335 38fb              412          ld       r3,p2dm
00000337 38fa              413          ld       r3,p2cm
00000339 38f9              414          ld       r3,p2bm
0000033b 38f8              415          ld       r3,p2am
0000033d 38f7              416          ld       r3,p4od
0000033f 38f6              417          ld       r3,p4d
00000341 38f5              418          ld       r3,h1c
00000343 38f4              419          ld       r3,h0c
00000345 38f1              420          ld       r3,pm
00000347 38d1              421          ld       r3,p1
00000349 38f0              422          ld       r3,p0m
0000034b 38ed              423          ld       r3,uie
0000034d 38ec              424          ld       r3,urc
0000034f 38eb              425          ld       r3,utc
00000351 38ea              426          ld       r3,sio
00000353 38e9              427          ld       r3,sie
00000355 38e8              428          ld       r3,srcb
00000357 38e7              429          ld       r3,srca
00000359 38e6              430          ld       r3,stc
0000035b c4e4c2            431          ldw      rr2,c1c
0000035e 38e5              432          ld       r3,c1cl
00000360 38e4              433          ld       r3,c1ch
00000362 c4e2c2            434          ldw      rr2,c0c
00000365 38e3              435          ld       r3,c0cl
00000367 38e4              436          ld       r3,c1ch
00000369 38e1              437          ld       r3,c1ct
```

```
0000036b 38e0                     ld       r3,c0ct

                        ; Bank l Special Registers

0000036d 38ff                     ld       r3,wumsk
0000036f 38fe                     ld       r3,wumch
00000371 38fb                     ld       r3,umb
00000373 38fa                     ld       r3,uma
00000375 c4f8c2                   ldw      rr2,ubg
00000378 38f9                     ld       r3,ubgl
0000037a 38f8                     ld       r3,ubgh
0000037c c4f0c2                   ldw      rr2,dc
0000037f 38fl                     ld       r3,dcl
00000381 38f0                     ld       r3,dch
00000383 c4eec2                   ldw      rr2,syn
00000386 38ef                     ld       r3,synh
00000388 38ee                     ld       r3,synl
0000038a 38ed                     ld       r3,smd
0000038c 38ec                     ld       r3,smc
0000038e 38eb                     ld       r3,smb
00000390 38ea                     ld       r3,sma
00000392 c4e8c2                   ldw      rr2,sbg
00000395 38e9                     ld       r3,sbgl
00000397 38e8                     ld       r3,sbgh
00000399 c4e4c2                   ldw      rr2,cltc
0000039c 38e5                     ld       r3,cltcl
0000039e 38e4                     ld       r3,cltch
000003a0 c4e2c2                   ldw      rr2,c0tc
000003a3 38e3                     ld       r3,c0tcl
000003a5 38e2                     ld       r3,c0tch
000003a7 38el                     ld       r3,clm
000003a9 38e0                     ld       r3,c0m

                        ;upper case test
000003ab 38de                     ld       r3,SYM
000003ad 38dd                     ld       r3,IMR
000003af 38dc                     ld       r3,IRR
000003bl c4dac2                   ldw      rr2,IP
000003b4 38db                     ld       r3,IPL
000003b6 38da                     ld       r3,IPH
000003b8 c4d8c2                   ldw      rr2,SP
000003bb 38d9                     ld       r3,SPL
000003bd 38d8                     ld       r3,SPH
000003bf 38d7                     ld       r3,RPl
000003cl 38d6                     ld       r3,RP0
000003c3 38d5                     ld       r3,FLAGS
000003c5 38d4                     ld       r3,P4
000003c7 38d3                     ld       r3,P3
000003c9 38d2                     ld       r3,P2
000003cb 38dl                     ld       r3,Pl
000003cd 38d0                     ld       r3,P0

                        ; Bank 0 Special Registers

000003cf 38ff                     ld       r3,IPR
```

```
000003d1 38fe         493          ld        r3,EMT
000003d3 38fd         494          ld        r3,P2BIP
000003d5 38fc         495          ld        r3,P2AIP
000003d7 38fb         496          ld        r3,P2DM
000003d9 38fa         497          ld        r3,P2CM
000003db 38f9         498          ld        r3,P2BM
000003dd 38f8         499          ld        r3,P2AM
000003df 38f7         500          ld        r3,P40D
000003e1 38f6         501          ld        r3,P4D
000003e3 38f5         502          ld        r3,H1C
000003e5 38f4         503          ld        r3,H0C
000003e7 38f1         504          ld        r3,PM
000003e9 38d1         505          ld        r3,P1
000003eb 38f0         506          ld        r3,P0M
000003ed 38ed         507          ld        r3,UIE
000003ef 38ec         508          ld        r3,URC
000003f1 38eb         509          ld        r3,UTC
000003f3 38ea         510          ld        r3,SIO
000003f5 38e9         511          ld        r3,SIE
000003f7 38e8         512          ld        r3,SRCB
000003f9 38e7         513          ld        r3,SRCA
000003fb 38e6         514          ld        r3,STC
000003fd c4e4c2       515          ldw       rr2,C1C
00000400 38e5         516          ld        r3,C1CL
00000402 38e4         517          ld        r3,C1CH
00000404 c4e2c2       518          ldw       rr2,C0C
00000407 38e3         519          ld        r3,C0CL
00000409 38e2         520          ld        r3,C0CH
0000040b 38e1         521          ld        r3,C1CT
0000040d 38e0         522          ld        r3,C0CT
                      523
                      524 ; Bank 1 Special Registers
                      525
0000040f 38ff         526          ld        r3,WUMSK
00000411 38fe         527          ld        r3,WUMCH
00000413 38fb         528          ld        r3,UMB
00000415 38fa         529          ld        r3,UMA
00000417 c4f8c2       530          ldw       rr2,UBG
0000041a 38f9         531          ld        r3,UBGL
0000041c 38f8         532          ld        r3,UBGH
0000041e c4f0c2       533          ldw       rr2,DC
00000421 38f1         534          ld        r3,DCL
00000423 38f0         535          ld        r3,DCH
00000425 c4eec2       536          ldw       rr2,SYN
00000428 38ef         537          ld        r3,SYNH
0000042a 38ee         538          ld        r3,SYNL
0000042c 38ed         539          ld        r3,SMD
0000042e 38ec         540          ld        r3,SMC
00000430 38eb         541          ld        r3,SMB
00000432 38ea         542          ld        r3,SMA
00000434 c4e8c2       543          ldw       rr2,SBG
00000437 38e9         544          ld        r3,SBGL
00000439 38e8         545          ld        r3,SBGH
0000043b c4e4c2       546          ldw       rr2,C1TC
0000043e 38e5         547          ld        r3,C1TCL
```

```
00000440 38e4               548         ld       r3,C1TCH
00000442 c4e2c2             549         ldw      rr2,COTC
00000445 38e3               550         ld       r3,COTCL
00000447 38e2               551         ld       r3,COTCH
00000449 38e1               552         ld       r3,C1M
0000044b 38e0               553         ld       r3,COM
                            554
                            555 ;test for condition codes
                            556
0000044d 0d0080             557         jp       f,128
                            558
00000450 6d0080             559         jp       z,128
00000453 ed0080             560         jp       nz,128
00000456 6d0080             561         jp       eq,128
00000459 ed0080             562         jp       ne,128
                            563
0000045c 7d0080             564         jp       c,128
0000045f fd0080             565         jp       nc,128
                            566
00000462 ad0080             567         jp       gt,128
00000465 1d0080             568         jp       lt,128
00000468 9d0080             569         jp       ge,128
0000046b 2d0080             570         jp       le,128
                            571
0000046e dd0080             572         jp       pl,128
00000471 5d0080             573         jp       mi,128
                            574
00000474 cd0080             575         jp       nov,128
00000477 4d0080             576         jp       ov,128
                            577
0000047a bd0080             578         jp       ugt,128
0000047d 7d0080             579         jp       ult,128
00000480 fd0080             580         jp       uge,128
00000483 3d0080             581         jp       ule,128
                            582
```

USER'S GUIDE



ZILOG

UNIVERSAL OBJECT FILE UTILITIES

II

Related Documents

Kernighan, Brian W. and Ritchie, Dennis M. The C Programming Language. Englewood Cliffs, NJ: Prentice-Hall, 1978.

IEEE Standard 695-1985. "The Microprocessor Universal Format for Object Modules."

Trademark Acknowledgements

UNIX is a trademark of AT&T Bell Laboratories; Zilog is licensed by AT&T Technologies, Inc.

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.

All specifications (parameters) are subject to change without notice. The applicable Zilog test documentation will specify which parameters are tested.

Zilog does not support the software mentioned in this publication, use at own risk.

ZILOG UNIVERSAL OBJECT FILE UTILITIES USER'S GUIDE
TABLE OF CONTENTS

CHAPTER TITLE AND SUBSECTIONS PAGE

CHAPTER 1: INTRODUCTION

CHAPTER 2: MCONV

CHAPTER 3: MDUMP

CHAPTER 4: MLIB

CHAPTER 5: MLINK

CHAPTER 6: MLIST

CHAPTER 7: MLIST

CHAPTER 8: MLORDER

Chapter Title and Subsections **Page**

Chapter 9: MMM

Chapter 10: PROTOCOL

Chapter 11: Other Programs

Appendices

Chapter 1
INTRODUCTION

1.1. OVERVIEW

1.1.1. Product Overview

The Universal Object File Utilities are part of Zilog's MUFOM-output cross-software family. The utilities allow the programmer to combine, display, and load machine-language object modules. The utilities are universal because they can process object modules produced by any of Zilog's MUFOM-outout cross-assemblers.

MUFOM is an acronym for Microprocessor Universal Format for Object Modules. MUFOM was developed by the IEEE as a format for representing machine-language object modules for any microprocessor. By using the MUFOM object format, Zilog supports all its assemblers (and compilers) using only one set of programs, the Universal Object File Utilities.

1.1.2. Manual Overview

This manual provides the following information:

- A brief description of the program's features.
- A complete definition of the command line syntax.
- A complete definition of the utilities' functions.
- Tutorials for the more complex portions of the utilities.
- A complete definition of the input file format.
- A complete definition of the output file format.

Section 1.2 briefly describes the utilities and their uses, and Section 1.3 describes how to invoke the utilities and the general command line syntax.

Chapters 2 through 10 discuss each utility in turn. Within each chapter command syntax, feature descriptions, and examples are provided. Chapter 11 describes three special-purpose programs which are also supplied with the utilities.

Appendix A provides a discussion of and specifications for the MUFOM object file format. Appendices B and C discuss the Tektronix Hex format and Intel Hex format, respectively. Appendix D lists the error messages.

Appendix E is the glossary. You do not need to understand the MUFOM object-file format to use these utilities. There are, however, a number of terms used when discussing MUFOM products that you should understand. These terms are defined in Appendix E. It is suggested that you familiarize yourself with these terms before continuing with the rest of this User's Guide.

1.2. UTILITIES DESCRIPTION

This section presents a brief description of each utility and its usage. Figure 1-1 shows how the utilities fit into the software development process.

1.2.1. mconv

mconv is an object format converter. It converts object modules from MUFOM ASCII format to MUFOM binary format and vice versa.

1.2.2. mdump

mdump is the object code dumper. It displays information about an object module, its sections, and its load data in human-readable form.

1.2.3. mlib

mlib is the object-code library maintenance utility. It allows object files to be combined into libraries which can be automatically searched by *mlink*.

1.2.4. mlink

mlink is a relocating linker. It accepts an arbitrary number of input files (limited only by available memory), resolves external references between files, combines file sections, and locates sections at absolute addresses. *mlink* also generates relocatable output modules which can be re-linked later.

1.2.5. mlist

mlist is the object code lister. It reconstructs an assembler-like listing from an object module, using special comments which are optionally inserted in the object module by the assembler.

1.2.6. mload

mload is a download format converter that translates MUFOM-format object modules into a form suitable for transmission (downloading) to development modules, emulators, or PROM programmers. The output formats supported are Tektronix Hex, Intel Hex, and a simplified form of MUFOM. *mload* is intended to be used with *protocol* or a similar communication program.

1.2.7. mlorder

Ix mlorder examines a set of object files to determine the optimum ordering for them in a library file, which can then be constructed using *mlib*.

1.2.8. mnm

mnm is the object module symbol lister. It displays information about the symbols within an object module.

1.2.9. protocol

protocol is a communication utility for transmitting files (typically load modules generated by mload) from a development host system to a target system (downloading) or vice versa (uploading). It supports a variety of handshakes to provide reliable transmission.

1.2.10. Other Programs

Three other programs are supplied with the Object File Utilities; they are intended for rather specialized purposes and will not be needed by most users.

1.2.10.1. mar

mar is an older version of *mlib*, producing an *archive* file which is compatible with the previous release of *mlink*.

1.2.10.2. m2a

m2a converts MUFOM files to *a.out* form, the object file format used on Zilog's S8000 microcomputers. This is provided for users of Zilog's EMS-8000 emulators, which use that format for downloading.

1.2.10.3. muimage.c

muimage.c is a MUFOM loader provided in *source* form for user customization.

Figure 1-1. The Universal Object File Utilities in the Software Development Process

1.3. UTILITY INVOCATION

This section describes the invocation of the object file utilities. The syntactic notations used in this section and throughout the rest of the manual are described below.

1.3.1. Syntactic Notation.

[item]
Square brackets indicate that the item is optional.

item1 | item2
A vertical bar indicates that either of the two items can be provided.

item ...
Three periods indicate that there can be one or more occurrences of the preceding item.

item *
An asterisk indicates that there can be zero or more occurrences of the preceding item.

N
N stands for a decimal number.

H
H stands for a hexadecimal number.

1.3.2. Command Invocation

Each utility is a separate program, invoked by using its name as a command. The command name is followed by zero or more "arguments" separated by spaces; command arguments may be filenames, numbers, or so-called "options".

Command line syntax follows the UNIX* convention in which a '-' sign followed by a one-character option identifier (with no intervening spaces) is parsed as an option (e.g., -o). Options can appear in any order. Case is not significant in option identifiers; they may be uppercase or lowercase letters.

Some options may be followed by a number or filename. A space is optional between the option letter and the number or filename, and required following it.

---

* UNIX is a trademark of AT & T Bell Laboratories; Zilog is licenced by AT&T Technologies, Inc.

Option characters may be concatenated (e.g., two options, such as -a and -b, can be written as -ab), provided the first option (-a in this example) does not expect to be followed by a number or filename.

For example, the command

```
    mlink -i foo.o -ofoo -rz
```

illustrates most of these principles: The -i and -o options are each followed by a filename (foo.o for -i, foo for -o. The two single-character options -r and -z are combined as -rz.

Chapter 2
MCONV

2.1. INTRODUCTION

The *mconv* utility is a filter that converts an object module from one format to another. MUFOM object files can be in either ASCII character or binary form. Object modules in binary form save space, while character form allows easy examination and reading by the user, and is more useful for downloading over serial links.

2.2. COMMAND SYNTAX AND OPTIONS

The *mconv* conversion utility is invoked by the following command:

```
        mconv [options] [file]
```

II

If no input file is specified, standard input is converted.

The command-line options are:

-b Convert the source to binary form.

-c Convert the source to character form; this is the default option.

-l Retain local symbols in the output. If this option is not supplied, only global and external symbols will be listed.

-k N Retain MUFOM comments up to level N in the output.

-o file
Direct output to the given file rather than to standard output.

Chapter 3
MDUMP

3.1. INTRODUCTION

The *mdump* utility is used to display MUFOM object code in a user-friendly format. It accepts MUFOM object modules as input and can output four items of information: the object module header, the section table, the link map, and the load data.

3.2. COMMAND SYNTAX AND OPTIONS

The command syntax for this utility is as follows:

```
   mdump [options] [file]
```

If no file is specified, then the standard input will be dumped.

The command-line options are:

-h   Display the header information.

-l   Display the load data.

-m   Display the link map.

-s   Display the section table.

If none of -h, -l, -m, or -s is given, all information is displayed.

-o file
   Direct output to the given file instead of standard output.

-k N
   Print the MUFOM comments within the object module with a level less than or equal to N. See Appendix A for a discussion of MUFOM comments.

3.3. DISPLAY FORMATS AND EXAMPLES

This section describes the formats of the four items of information in *mdump's* output. They may be individually selected for display by command-line options; by default all four items are output.

3.3.1. The Header

The first part of *mdump's* output is a header containing general information about the module. The header information includes:

- Module name
- Target processor
- Character/Binary format
- Address length and byte order
- Creation date and time
- Absolute/Relocatable
- Entry point
- Program size (in hex and decimal)

A typical module header is shown below:

```
    Module: test; target Z80K; character form.
        Address length 4 bytes; MSB first.
        Created 1986/04/02 09:39:38.
        Entry point = 00000001.
        Total size =      e58 (3672); absolute.
```

3.3.2. The Section Table

Following the header, mdump lists a table of all the sections in the object module, as shown in the two examples below. Note that some fields may be blank if no values have been set for them. In particular, the LOCATION field is blank for relocatable sections.

```
 SECN LOCATION --SIZE-- --ALIGN- --PAGE-- NAME:ATTS
    0          00000d4a 00000002          :
    1          00000014 00000002          sec1_name:
    2          0000000a 00000002          sec2:XP
    3          00000002 00000004          sec3:
    4          0000000c 0000000c          code:X
    5          00000006 00000002          data:
    6          00000010 00000002          bss:BCW
    7 00001000 00000014 00000002          abs:A
    8          00000002 00000002          comm:M
```

```
 SECN LOCATION --SIZE-- --ALIGN- --PAGE-- NAME:ATTS
    0 00000000 00000166 00000002 00010000 allfoo:ANSW
    1 00005000 0000092e                   libcode:ANSX
    2 00005000 00000060 00000002 00010000 code:ANSX
    3 00005060 000000f0 00000002 00000000 Ccommon:ABNSW
```

The SECN column displays the section number. Each section has a number associated with it that differentiates it from the other sections in the object module. The LOCATION column displays starting address (lower boundary) of the section. If the section is relocatable then the LOCATION column's entry will be blank. The SIZE column shows the size in hexadecimal of the section. The ALIGN and PAGE columns show the alignment boundary and page size of the section, if defined.

The NAME:ATTS column shows the name and attributes of the section, separated by a colon. See Section 5.2.2.4 in the chapter on mlink for a discussion of section attributes and their meanings.

3.3.3. The Link Map

Object modules that are output by the linker, mlink, contain information about the files and sections that were linked together to form them. This information is called the Link Map, and is identical to that displayed by the -v option of mlink.

If no link map is present, mdump displays the message

```
        No link map information.
```

An example of a link map is shown below:

```
LINK MAP:  Input Sections
FILE  test.o                                created 1986/03/24 09:40:36
    0    1 L=00000006 S=00000d4a test.o,:ANSW
    1    5 L=00000a7e S=00000014 test.o,sec1_name:ANSW
    2    4 L=00000a74 S=0000000a test.o,sec2:APSX
    3    6 L=00000d94 S=00000002 test.o,sec3:ANSW
    4    3 L=00000d68 S=0000000c test.o,code:ANSX
    5    0 L=00000000 S=00000006 test.o,data:ANSW
    6    a L=000010bc S=00000010 test.o,bss:ABCNW
    7    7 L=00001000 S=00000014 test.o,abs:ANSW
    8    8 L=00001014 S=00000002 test.o,comm:AMNW
FILE  txxx.o                                created 1985/10/31 14:53:08
    0    2 L=00000d50 S=00000010 txxx.o,:ANSW
    1    8 L=00001014 S=00000004 txxx.o,comm:AMNW
```

Note that the link map includes the name and creation date of each file that was linked; if the file came from a library, the library name follows the filename in parentheses.

The line for each input file is followed by a line for each section that the file contains; the first two columns are the input and output section numbers, respectively.

For relocatable sections, "L=" is replaced by "R=", and the associated location is the offset of the input section within the (possibly larger) output section.

3.3.4. The Load Data

The Load Data is the data and code that will actually be loaded into the target machine's memory.

The data is displayed in the format shown below:

```
    Section number
    address:  --------object code--------   |ASCII equivalent|
    address:  --------object code--------   |ASCII equivalent|
    etc.
```

The load data is broken up into lines for display, each of which can show up to sixteen bytes of data. The display lines are aligned on modulo-16 byte boundaries with the address being the address of the first byte actually displayed. If the section is relocatable then the address is relative to the beginning of the section. If the section is absolute, then the address is the actual position in memory.

Example:

```
00000000  5e 08 50 54 a1 7e a1 6d a1 5c 8d c4 5e 0e 50 16   |^ PT ~ m \
00000010  83 22 5e 08 50 4a ab c0 0b 0c 00 00 5e 02 50 3c   | "^ PJ
00000020  20 e0 0a d0 5e 0e 50 3c 0c e4 5e 0e 50 34 83 22   |     ^ P<   ^
00000030  5e 08 50 4a a9 e0 a9 d0 5e 08 50 16 20 ea b1 20   |^ PJ      ^ P
00000040  20 d8 b1 00 83 02 5e 08 50 4a 5c f1 0c 02 00 00   |          ^ PJ\
00000050  a9 f5 9e 08 ab f5 5c f9 0c 02 00 00 5e 08 50 04   |         \
```

3.3.5. Disjoint Sections

It is important to note that the MUFOM format allows the object code for a section to be broken up into physically disjoint pieces. If pieces of sections are distributed randomly throughout the object module, mdump will not be able to display each section contiguously.
Instead, mdump will display the pieces of the sections as it receives them from the input file. The example below shows the load data of a module with two sections, each split into two pieces.

```
Section 0
00000000  61 62 63 64                                        |abcd
Section 1
00000000  30 32 33 34 35                                     |02345
Section 0
00000004              65 66 67 68                            |    efgh
Section 1
00000005                 36 37 38 39                         |     6789
```

Sections can also contain gaps (caused by assembler statements that reserve space without initializing it). Short gaps are represented by "......" within a single line; long gaps by "...." in the address field, as shown below:

```
    00000000  01 02 03 ........... 04 05 06
    ........
    0000006e                                           08 09
```

3.3.6. Displaying Relocation Information

Within MUFOM relocatable object code, references to unresolved external symbols and to locations in relocatable sections are represented as expressions. The formats used for displaying expressions are shown below:

Rnn+offset	relocatable address
Xnn+offset	external referance
**********	other expressions

The expressions are padded with periods to occupy the appropriate amount of space (three columns per byte). If there is insufficient space for the whole expression, it is abbreviated to its first letter and padded with periods. The following example illustrates all three formats.

```
    C0000000  R1+1234.... X0+1234.... *..........
```

If necessary, more detail about an expression can be obtained by running **mlist**, which can expand expressions completely.

Chapter 4
MLIB

4.1. INTRODUCTION

The *mlib* utility is used for creating and maintaining libraries of object modules for use with *mlink*. Libraries are stored in a form that permits efficient searching and linking of the modules they contain.

4.2. COMMAND SYNTAX AND OPTIONS

The *mlib* conversion utility is invoked by the following command:

```
mlib key lfile [name]...
```

Key is one character from the set "drtqxf" optionally followed by "v". *lfile* is the library file; the *names* are the constituent files in the library.

Note that a "key" is not an "option"; it has no leading "-" character. The meanings of the key characters are:

d Delete the named files from the library file.

r Replace the named files in the library file.

q Quickly append the named files to the end of the library file, without checking whether they are already in the library.

t Print a table of contents of the library file. If no names are given, all files in the library are tabled.

x Extract the named files from the library. If no names are given, all files in the library are extracted. The library file itself is not altered. The extracted files are put into the current working directory.

v Verbose. Gives more information about what mlib is doing. With *t* this includes a listing of the symbols in each module as well as the names of the modules.

f The first and only "name" in the command line is the name of a file which contains the list of filenames.

4.3. EXAMPLES

To combine several files (say, "file1.o", "file2.o" and "file3.o") into a library, use the command:

```
        mlib q foo.lib file1.o file2.o file3.o
```

If one of the files is modified, it can be replaced with the command:

```
        mlib r foo.lib file2.o
```

To add another file to the library, use

```
        mlib q foo.lib file4.o
```

To find out what is in the library, use

```
        mlib t foo.lib
```

To break the library into separate files, use the command

```
        mlib x foo.lib
```

Note that the library file is unaffected by this operation. A single file can be extracted with the command

```
        mlib x foo.lib file2.o
```

If a filename containing a list of files, say "bar", has been prepared (for example as the output of *mlorder*), we can use it to create a library with the command

```
        mlib qf bar.lib bar
```

Chapter 5
MLINK

5.1. INTRODUCTION

The *mlink* utility is used to assign absolute addresses to relocatable sections in MUFOM input modules, and to combine (link) two or more separate object modules into one module. Linking allows programs to be developed as groups of smaller, easier-to-manage modules that can then be combined to form a single object module. All of a program's modules can be merged at one time or they can be combined into submodules (sometimes called pre-links) which can themselves be combined in a subsequent *mlink* run.

5.1.1. Modules And Sections

In order to understand the linking process, it is useful to understand the way in which MUFOM files are constructed. (The following discussion is a shortened version of that found in Sections A.2 and A.3 of the Appendix on MUFOM. See the Appendix for more detail.) MUFOM object files are divided into *sections* each of which is destined to be loaded into a separate area of memory. Each section has a *name*, a *size*, *attributes*, and (if not relocatable) a *location*. Each section also has a *section number* which is used to refer to it internally. In Zilog's implementation these section numbers correspond to the order of the sections in the section table. (See Section A.2 in the Appendix on MUFOM for a discussion of the various section attributes and their meanings.)

It is important to note that the name of a section may be null (in which case the section is referred to as "unnamed"), and that the names of sections *need not be unique*. Thus, a file may contain several sections named "code". The advantage of this is that the linker can relocate such sections separately; thus on a Z8001 not all "code" sections have to be in the same segment.

Sections can be referenced in the linker by either their name, their attributes, or the name of the file from which they came.

MUFOM object files as implemented at Zilog are divided into three regions: a *section table* giving all information about the file's sections except their actual contents, a *symbol table* which defines the N, I, and X variables which represent local, internal, and external symbols respectively, and finally the *load data*, the LD and LR commands which define the actual contents of the sections. These regions are delimited by special MUFOM comment commands; separating them in this way makes the linker and other utilities run faster.

As implemented at Zilog, the N, I, and X variables of any object file are allocated contiguously starting from N0, I0, and X0. The variable indexes do *not*, however, necessarily correspond to the order of the variables in the symbol table. It is only guaranteed that there will be no gaps in the numbering.

MUFOM permits comments (CO commands) in object files; Zilog's assemblers use level 0 comments for error messages, level 1 comments for compiler-supplied debugging information, level 2 comments for assembler source lines, and level 3 for assembler line numbers and formatting information. This permits debuggers and other utilities (such as *mlist*) to reconstruct the source from the object file.

The comments that introduce the section table, symbol table, and section contents have levels 100(hex), 101, and 102 respectively.

5.1.2. The Link Process

The command arguments are parsed from left to right. Each argument is essentially a command to the linker.

The linker maintains two lists of sections: the Input List and the Output List. The **-i file...** command-line argument gets sections from input files and puts them into the Input List. As each file is input, its section table is processed to construct entries on the Input List, and its symbol table is processed to resolve external references.

The **-s** command-line argument selects sections from the Input List and puts them into the Output List at the Current Location. As each section is selected it is assigned a starting location, and the Current Location is incremented by the length of the section*.

----

* Things are actually a little more complicated; assignment of location is deferred until either a **-n** or **-o** option is encountered. This is done to allow the **-u** option to "unselect" sections. Also, if an absolute section is encountered, the Current Location for the next section will be the location of the absolute section plus its size.

The -o **file** command-line argument appends the sections in the Output List to a file. The output list is then cleared. when the code or data contained in a section is output to a file, the values of external or relocatable references and link-time expressions are substituted.

All other arguments operate on the Output List or the Symbol Table.

After the command line is parsed, the linker makes two passes over the input files. In the first pass, the symbol and section information in each input file is read and processed, and an Output List is constructed for each output file. With each -o argument, locations are assigned to relocatable sections.

At the end of the first pass, any remaining sections are put on the Output List of the last file mentioned, locations are assigned to common symbols, and still-undefined externals are identified.

In the second pass the output files are written. For each output file, symbols and program data are copied from the input files. Link-time expressions (including relocation) and external references are replaced by their values during the copying process.

5.2. COMMAND LINE SYNTAX AND OPTIONS

The command-line options for *mlink* are given below in Table 5-1. More complete discussions of each option are given in the following sections.

The command line is processed from left to right; each option with its sub-arguments is essentially a command to the linker. Unlike most of the other utilities, the order of the command-line arguments is significant in *mlink*.

Table 5-1. mlink Options Summary

Option	Description
Input and Output File Options	
-i [ifile]*	specify input files
-o [ofile] [isection]*	specify output file
Section Options	
-s [isection]*	select input sections
-n [osection]	name and combine sections
-address	set location for next section
-t address	set top loc. for previous sect.
-r	relocatable sections follow
-m N	mark loc./return to mark
-u [isection]*	unselect sections
Output File Options	
-b	binary format output
-c	character format output
-k N	keep comments in output
Symbol Options	
-l	discard local symbols
-d	define C common symbols
-x [sym_op]*	process external symbols
-g [sym_op]*	process global symbols
-e [value]	specify entry point
Other Options	
-p	proceed even if errors
-v [N]	set verbosity level
-w	suppress warnings
-z	Z8000 segments
-f file	command file input

5.2.1. Input and Output File Options

The Input and Output options specify the input and output files for the link operation. If no output files are specified, output goes by default to "m.out". Note that more than one output file can be specified.

If no input files are specified, mlink will generate an output file containing no load data. This can be useful if the symbol options are used to define symbols. Also, the section options can be used to create empty sections with specified names, attributes, and locations.

5.2.1.1. File Option Syntax

```
        file_opt   ::= -i [ifile]*
                    |  -o [ofile] [isection]*

        ifile      ::=  object_filename | archive_filename
        ofile      ::=  object_filename
```

5.2.1.2. File Option Descriptions

-i [ifile]*
Input the specified files, putting their sections into the Input List. As each file is processed, its sections are placed into the Input List in numerical order.

A -i is assumed at the beginning of the command, so the following are equivalent:

```
mlink   -i file1.o
mlink   file1.o
```

If a library file is specified, it is searched for modules containing global symbols that match undefined externals currently in the Symbol Table. If any such modules are found, they are added to the Input List.

If searching a library causes any new externals to be added to the symbol table, it is searched again.

-o [ofile] [isection]*
Appends the Output List to the given file. If no file is given, the sections in the Output List are thrown away (but space is still allocated for them). Note that more than one output file can be specified; this

feature can be used for loading into different segments or PROMs, or for constructing overlays.

If section specifiers are given, only those specified sections (in addition to the sections in the file's Output List) are included in the output file's section table. Filenames in the section specifiers refer to output files. This feature is used to ensure that overlays do not reference sections in mutually exclusive overlays.

5.2.1.3. Automatic Section Combining

Some section attributes specify that sections are to be combined automatically in various ways. (See Section A.2.2 for the discussion of Overlap attributes and their effects.) Such sections are combined when they are first encountered in **-i** (input) file lists, and only the sections in the current Input List are looked at to find sections to combine with. Thus, if a **-i** option comes after some sections have been selected with a **-s** option, the sections that have already been selected will not be combined with, even if their names and attributes match those of some new sections. This provides a way to override the automatic section-combining mechanism.

5.2.2. Section Options

The section options allow you to specify explicitly how the sections input object modules are selected and positioned in the output modules. Sections in the input modules are kept in an internal structure called the Input List until selected by a **-s** (select) option. They are then moved to the Output List. Sections on the Output List are moved into an output file when a **-o** (output) option is encountered.

5.2.2.1. Section Option Syntax

```
        sec_opt    ::= -s [isection]*
                    |  -n [osection]
                    |  -address
                    |  -t address
                    |  -r
                    |  -m N
                    |  -u [isection]*

        address    ::=  digit [hexdigit]*
        isection   ::=  [filename,][sec_name][:att_match]
        osection   ::=  [sec_name][:attributes]

        att_match  ::=  [att_term] [+att_term]*
        att_group  ::=  [letter | -letter]...
        sec_name   ::=  symbol | +
        attributes ::=  letter*
```

5.2.2.2. Section Option Descriptions

-s [isection]*

Select sections from the Input List and put them into the Output List. They will be located starting at the Current Location, which is initially zero. Sections matching the first "isection" in the select list will be put into the output list first; sections that match the same "isection" will stay in the same order that they had in the Input List. The section selectors are described in more detail below.

If no sections are specified the entire Input List is selected, except for Postpone sections (sections with the "P" attribute.) If Postpone sections are selected in other cases, they are placed after all the other sections in the same selection.

-n [osection]

Combines all the sections currently in the Output List and gives them the given name (and attributes, if specified). If no section or ":attributes" is specified, the combined section is unnamed. If no attributes are specified, the new section has the default attributes (:WSN). No attributes are inherited from any of the constituent sections.

-address

Sets the Current Location to the given address.

-t address

Adjusts the base address of the last section in the Output List (i.e., the last section selected before the -t option) so that its top comes as close as possible to the given address without violating its alignment constraint.

-r

Any sections selected after the -r argument will be relocatable (until the next -address or -t argument).

-m N

If this is the first time the given mark number is encountered, set that mark to the Current Location. Otherwise, set the Current Location to the value of the given mark. This argument is used for aligning overlays.

-u [section]*

"Unselect" the given sections, moving them from the output list back into the input list. This can be used, for example, to select "all but" a given section, or to construct a "Postpone" section which will be output later.

5.2.2.3. Section Selectors

A section selector as used in the -s option has three components: a filename, a section name, and an attribute-match specifier. Any of these may be omitted, in which case all sections matching the other components are selected (the limiting case is -s with no section selectors, which selects all sections in the Input List).

The format of a section selector is

```
filename,secname:attributes
```

Note that no spaces are permitted between the fields. Note also that the filename must not contain a comma (this is permitted only in UNIX, and is rare in any case).

File Name

The file name component of a section selector refers to the input file from which the section came. It is terminated by a comma.

Section Name

The section name component of a section refers to the name of the section as given in a MUFOM 'ST' command in the input module from which it came. Sections can be un-named; such sections can be selected explicitly by using '+' as the section name component of a section selector.

Attribute Match

The attribute-match component of a section refers to the attributes as given in a MUFOM 'ST' command in the input module from which it came. Because sections have more than one attribute, the attribute-match component can be rather complicated.

The attribute-match component of a section selector starts with a colon, and consists of zero or more "terms" separated by '+' signs. A section matches the attribute-match if it matches *any one* of the fields. Thus, '+' has the meaning of "logical OR".

A term in an attribute-match consists of one or more letters (case is not significant), each optionally preceded by a '-' sign. If a letter is non-negated in a term, the corresponding attribute *must be present* in a section in order for it to match. If a letter is negated in a term, the corresponding attribute *must not be present* in a section in order for it to match. Thus, letters (attributes) in a term are connected by "logical AND", and '-' has the meaning of "logical NOT".

Note that the attributes matched by a term may be a *subset* of the attributes which a section actually has.

See Section A.2 in the Appendix on MUFOM for a discussion of the various section attributes and their meanings.

5.2.2.4. Default Selection

At the end of the link, any sections still in the Input List are selected, and the Output List is appended to the output list of the last file mentioned in a -o argument (as if the -o argument had been moved to the end of the command). If no -o argument is given, the default filename is *m.lnk*.

The sections are selected in the sequence: code (X attribute), read-only data (R attribute), other non-BSS data, C common and BSS (B attribute). The default behavior is thus equivalent to

```
     mlink -o m.lnk <actual arguments> -s :X :R :-B -d -s :B
```

5.2.3. Output File Options

The following arguments apply to the next output file, or to the last output file if they follow the last -o argument.

5.2.3.1. Output File Option Syntax

```
        ofile_opt   ::= -b
                        -c
                        -k N
```

5.2.3.2. Output File Option Descriptions

The following options apply to the output file specified by the next -o argument, or if they follow the last -o argument, to the last output file specified.

-b

Put out the next output file in binary form.

-c

Put out the next output file in character form.

-k

In the next output file, keep comments up to and including level N. To retain source information for use with *mlist*, use -k3.

5.2.4. Symbol Options

The symbol options operate on the symbol table which is generated in the linking process. They allow new symbols to be defined, or sets of symbols to be excluded from the output symbol table.

5.2.4.1. Symbol Option Syntax

```
sym_opt     ::= -l
             |  -d
             |  -x [sym_op]*
             |  -g [sym_op]*
             |  -e [value]

symbol      ::=  letter[letter|digit]*
value       ::=  [symbol | address]

sym_op      ::=  symbol
                 + symbol
                 + symbol = [value]
                 + symbol length
```

5.2.4.2. Symbol Option Descriptions

-l

Do not put local symbols in the next output file.

-d

Define a section for any common symbols encountered up to this point, and select it into the Output List. Common symbols are used by the C compiler and other compilers to hold un-initialized data. The common section defined has the name "Ccommon" and the attributes "BNSW".

-x [sym_op]*

-g [sym_op]*

Process external or global symbols. With no sym_op's given, the default operation is to strip the symbols from *all* output files. "Stripped" symbols are not actually removed from the internal symbol table, but are marked so that they will not be output. The operations are:

symbol

Strip a particular symbol.

+ symbol

Add a particular symbol. Externals are added as undefined, globals as zero.

+ symbol = [value]

Add a symbol with the given value. If the value is omitted, the Current Location is used. In the -x argument, a Weak External is constructed. A Weak External is an external symbol which receives the given value as a default if no corresponding global symbol is defined in the link.

+ symbol length
Add a C-type Common symbol with the given length. Used in the -x argument only.

-e [value]
Set the Entry Point to the given value (symbol or address). If no value is given, the default is the Current Location. If no -e argument is given, the default is the entry point of the first input file that has one. If no input file has an entry point, it is up to the loader or operating system to define one.

5.2.5. Other Options

The following arguments are non-positional, and apply to the entire link operation.

5.2.5.1. Other Option Syntax

```
other_opt  ::= -p
             |  -v [number]
             |  -w
             |  -z
             |  -f file
```

5.2.5.2. Other Option Descriptions

-v[n]
"Verbose": print information on Standard Error about what the linker is doing. The optional number selects different levels of information:

1 (default) Output a link map on Standard Error at the end of pass 1.

2 Output the name of each input and output file as it is opened.

3 Output information about each section as it is defined or selected.

4 Output more information about input file format errors.

-p
Proceed in spite of errors.

-w

Suppress warning messages.

-z

Perform Z8001-type segmented address arithmetic. With this option in effect, the next address after 100FFFF hex is 2000000; in other words bits 16-23 are not part of the address.

-f file

Take arguments from a file. Newlines in the file are treated as spaces. The file is effectively inserted into the command line in place of the -f argument. The file can contain comments starting with a semicolon (";") character and terminated by end of line.

5.3. CONSTRAINTS

All of *mlink's* tables are dynamically allocated, so that the number of symbols, sections, and files that can be handled depends mainly on the amount of memory available. In addition, Zilog's implementation of MUFOM imposes the following limits:

Symbol and Section Names: 127 characters.

Sections: 65536.

Local Symbols: 65536.

Global and External Symbols: 65536 total.

5.4. USING mlink: SOME EXAMPLES

This section describes the usage of the *mlink* utility through several examples.

5.4.1. The Sample Input Files

For the purposes of most of the following examples, we will use two input files called "file1.o" and "file2.o" with a structure similar to that produced by the C compiler. Each has three sections, called "code", "data", and "bss". ("bss" stands for "Block Started by Symbol", and is used for uninitialized data that is cleared to zero when the program is started.)

In addition, we will assume that "file1.o" contains a section called "rom" containing read-only data, that "file2.o" has an additional section called "stack" for the program's stack, and that both files contain some *common* symbols. (Common symbols are external symbols which are allocated in a BSS section if no corresponding global symbol is defined. They are used by C for uninitialized variables.)

The sections of the sample input files are shown below in tabular form (prepared by *mdump*), and graphically in Figure 5-1.

```
file1:
 SECN LOCATION --SIZE-- --ALIGN- --PAGE-- NAME:ATTS
    0          00000242 00000002          code:X
    1          00000231 00000002          rom:R
    2          00000232 00000002          data:
    3          00000200 00000002          bss:BCW

file2:
 SECN LOCATION --SIZE-- --ALIGN- --PAGE-- NAME:ATTS
    0          00000242 00000002          code:X
    1          00000234 00000002          data:
    2          00000200 00000002          bss:BCW
    3          00002000 00000002          stack:BP
 file1.o:
*-----------------------*
|    file1.o/code:X     |
+-----------------------+
|    file1.o/rom:R      |
+-----------------------+
|    file1.o/data:W     |
+-----------------------+
|    file1.o/bss:WBC    |
*-----------------------*

 file2.o:
*-----------------------*
|    file2.o/code:X     |
+-----------------------+
|    fila2.o/data:W     |
+-----------------------+
|    file2.o/bss:WBC    |
+-----------------------+
|    file2.o/stack:WBP  |
*-----------------------*
```

Figure 5-1. Example Input Files

5.4.2. Default Section Ordering

The simplest thing to do with our two sample files is to let the linker select sections in their default ordering, and locate them consecutively starting at zero. The linker's defaults are designed to "do the right thing" for C compiler output running in an environment like UNIX*. Thus, code (sections marked **executable** with the "X" attribute) is placed starting at address zero, followed by read-only data ("R" attribute), read-write initialized data ("W" attribute), BSS ("W" and "B" attributes), and stack ("W", "B", "P" attributes). The command for doing this is

```
        mlink -i file1.o file2.o -o ex1
```

where **ex1** is the name of the file which will receive the linker's output. Note that the **-i** option flag is not required, since it comes at the beginning of the command, and that if the **-o linked.out** option is omitted the output file will be called "m.lnk".

The resulting file's structure is shown below and in Figure 5-2.

Note that the two BSS sections have been combined automatically, because they have the "C" attribute. Sections with this attribute are automatically combined if their names and other attributes are the same. Also note that the section "Ccommon" has been created to hold the *common* symbols, and that the linker has filled in the default section attributes "W" ("writable"), "N" ("now", the inverse of "P") and "S" ("separate", the inverse of "C") wherever appropriate. Since the linker has given each section a location, they have also acquired the "A" ("absolute") attribute as well.

```
mlink -i file1.o file2.o -o ex1 -v

mlink v. 2.1 -- Zilog MUFOM linking utility

MAP:   ISecn OSecn  Location    Size      IFile/Name:Atts

  Output file ex1:
            0     0 L=00000000  S=00000242 file1.o/code:ANSX
            0     1 L=00000242  S=00000242 file2.o/code:ANSX
            1     2 L=00000484  S=00000231 file1.o/rom:ANRS
            2     3 L=000006b6  S=00000232 file1.o/data:ANSW
            1     4 L=000008e8  S=00000234 file2.o/data:ANSW
            -     5 L=00000b1c  S=00000032 /Ccommon:ABNSW
            -     6 L=00000b4e  S=00000400 /bss:ABCNW
            3     7 L=00000f4e  S=00002000 file2.o/stack:ABPSW

  Input files:
            0     0 L=00000000  S=00000242 file1.o/code:ANSX
            1     2 L=00000484  S=00000231 file1.o/rom:ANRS
            2     3 L=000006b6  S=00000232 file1.o/data:ANSW
            3     6 L=00000b4e  S=00000200 file1.o/bss:ABCNW
            0     1 L=00000242  S=00000242 file2.o/code:ANSX
            1     4 L=000008e8  S=00000234 file2.o/data:ANSW
            2     6 L=00000d4e  S=00000200 file2.o/bss:ABCNW
            3     7 L=00000f4e  S=00002000 file2.o/stack:ABPSW
```

```
file  ex1:
         *-----------------------*
         |    file1.o/code:X     |
         +-----------------------+
         |    file2.o/code:X     |
         +-----------------------+
         |    file1.o/rom:R      |
         +-----------------------+
         |    file1.o/data:W     |
         +-----------------------+
         |    file2.o/data:W     |
         +-----------------------+
         |  Ccommon:WB           |
         +-----------------------+
         |  bss:WBC              |
         |    file1.o/bss:WBC    |
         |    file2.o/bss:WBC    |
         +-----------------------+
         |    file2.o/stack:WBP  |
         *-----------------------*
```

Figure 5-2. Default Selection Ordering

The -v command-line option of *mlink* was used to generate the link map above; note that its format is slightly different from the information displayed by *mdump*. More information about what *mlink* is doing can be displayed with the -v3 option, as shown below:

```
mlink -i file1.o file2.o -o ex1 -v3

mlink v. 2.1 -- Zilog MUFOM linking utility
  Input file 'file1.o'
           0    - R=00000000 S=00000242 file1.o/code:NSX
           1    - R=00000000 S=00000231 file1.o/rom:NRS
           2    - R=00000000 S=00000232 file1.o/data:NSW
           3    - R=00000000 S=00000200 file1.o/bss:BCNW
  Input file 'file2.o'
           0    - R=00000000 S=00000242 file2.o/code:NSX
           1    - R=00000000 S=00000234 file2.o/data:NSW
           2    - R=00000200 S=00000200 file2.o/bss:BCNW
           3    - R=00000000 S=00002000 file2.o/stack:BPSW
  Select :X
           0    - R=00000000 S=00000242 file1.o/code:NSX
           0    - R=00000000 S=00000242 file2.o/code:NSX
  Select :R
           1    - R=00000000 S=00000231 file1.o/rom:NRS
  Select :-B
           2    - R=00000000 S=00000232 file1.o/data:NSW
           1    - R=00000000 S=00000234 file2.o/data:NSW
  Select
           -    - R=00000000 S=00000400 /bss:BCNW
           3    - R=00000000 S=00002000 file2.o/stack:BPSW

MAP:   ISecn OSecn  Location   Size     IFile/Name:Atts

  Output file ex1:
           0    0 L=00000000 S=00000242 file1.o/code:ANSX
           0    1 L=00000242 S=00000242 file2.o/code:ANSX
           1    2 L=00000484 S=00000231 file1.o/rom:ANRS
           2    3 L=000006b6 S=00000232 file1.o/data:ANSW
           1    4 L=000008e8 S=00000234 file2.o/data:ANSW
           -    5 L=00000b1c S=00000032 /Ccommon:ABNSW
           -    6 L=00000b4e S=00000400 /bss:ABCNW
           3    7 L=00000f4e S=00002000 file2.o/stack:ABPSW

  Input files:
           0    0 L=00000000 S=00000242 file1.o/code:ANSX
           1    2 L=00000484 S=00000231 file1.o/rom:ANRS
           2    3 L=000006b6 S=00000232 file1.o/data:ANSW
           3    6 L=00000b4e S=00000200 file1.o/bss:ABCNW
           0    1 L=00000242 S=00000242 file2.o/code:ANSX
           1    4 L=000008e8 S=00000234 file2.o/data:ANSW
           2    6 L=00000d4e S=00000200 file2.o/bss:ABCNW
           3    7 L=00000f4e S=00002000 file2.o/stack:ABPSW
  Output file 'ex1'
  -- Input file 'file1.o'
  -- Input file 'file2.o'
```

5.4.3. Selecting Sections by Name

Sometimes it is necessary to put the sections of the output file in some order other than *mlink*'s default ordering. (This is usually done in order to specify the addresses of the sections, as we will see in later examples, but is also useful for constructing large tables from data in several different modules.) Sections can be selected according to their name, their attributes, or their file of origin, or any combination of these.

The first example in this series will select sections by name, since the command line for doing this is somewhat simpler. For example, suppose you want the data sections to come first, followed by ROM, and then code, BSS, and stack in their usual order. The command for this and the resulting map are shown below.

```
mlink -i file1.o file2.o -s data rom -o ex2 -v

mlink v. 2.1 -- Zilog MUFOM linking utility

MAP:   ISecn OSecn  Location    Size       IFile/Name:Atts

  Output file ex2:
             2    0 L=00000000 S=00000232 file1.o/data:ANSW
             1    1 L=00000232 S=00000234 file2.o/data:ANSW
             1    2 L=00000466 S=00000231 file1.o/rom:ANRS
             0    3 L=00000698 S=00000242 file1.o/code:ANSX
             0    4 L=000008da S=00000242 file2.o/code:ANSX
             -    5 L=00000b1c S=00000032 /Ccommon:ABNSW
             -    6 L=00000b4e S=00000400 /bss:ABCNW
             3    7 L=00000f4e S=00002000 file2.o/stack:ABPSW

  Input files:
             0    3 L=00000698 S=00000242 file1.o/code:ANSX
             1    2 L=00000466 S=00000231 file1.o/rom:ANRS
             2    0 L=00000000 S=00000232 file1.o/data:ANSW
             3    6 L=00000b4e S=00000200 file1.o/bss:ABCNW
             0    4 L=000008da S=00000242 file2.o/code:ANSX
             1    1 L=00000232 S=00000234 file2.o/data:ANSW
             2    6 L=00000d4e S=00000200 file2.o/bss:ABCNW
             3    7 L=00000f4e S=00002000 file2.o/stack:ABPSW
```

```
file  ex2:
        *-----------------------*
        |    file1.o,data:W     |
        +-----------------------+
        |    file2.o,data:W     |
        +-----------------------+
        |    file1.o,rom:R      |
        +-----------------------+
        |    file1.o,code:X     |
        +-----------------------+
        |    file2.o,code:X     |
        +-----------------------+
        |  Ccommon:WB           |
        +-----------------------+
        |  bss:WBC              |
        |    file1.o,bss:WBC    |
        |    file2.o,bss:WBC    |
        +-----------------------+
        |    file2.o,stack:WBP  |
        *-----------------------*
```

Figure 5-3. Selecting Sections by Name

5.4.4. Selecting Sections by File

Selecting sections according to their file of origin is equally simple. The syntax for a filename selector is the filename followed by a comma, as in the following example where we select all the sections in "file1.o" followed by all the sections in "file2.o".

```
mlink -i file1.o file2.o -s file1.o/ file2.o/ -o ex3 -v

mlink v. 2.1 -- Zilog MUFOM linking utility

MAP:    ISecn OSecn  Location   Size     IFile/Name:Atts

  Output file ex3:
              0    0 L=00000000 S=00000242 file1.o/code:ANSX
              1    1 L=00000242 S=00000231 file1.o/rom:ANRS
              2    2 L=00000474 S=00000232 file1.o/data:ANSW
              0    3 L=000006a6 S=00000242 file2.o/code:ANSX
              1    4 L=000008e8 S=00000234 file2.o/data:ANSW
              3    5 L=00000b1c S=00002000 file2.o/stack:ABPSW
              -    6 L=00002b1c S=00000032 /Ccommon:ABNSW
              -    7 L=00002b4e S=00000400 /bss:ABCNW

  Input files:
              0    0 L=00000000 S=00000242 file1.o/code:ANSX
              1    1 L=00000242 S=00000231 file1.o/rom:ANRS
              2    2 L=00000474 S=00000232 file1.o/data:ANSW
              3    7 L=00002b4e S=00000200 file1.o/bss:ABCNW
              0    3 L=000006a6 S=00000242 file2.o/code:ANSX
              1    4 L=000008e8 S=00000234 file2.o/data:ANSW
              2    7 L=00002d4e S=00000200 file2.o/bss:ABCNW
              3    5 L=00000b1c S=00002000 file2.o/stack:ABPSW


              file  ex3:
                   *------------------------*
                   |     file1.o/code:X     |
                   +------------------------+
                   |     file1.o/rom:R      |
                   +------------------------+
                   |     file1.o/data:W     |
                   +------------------------+
                   |     file2.o/code:X     |
                   +------------------------+
                   |     file2.o/data:W     |
                   +------------------------+
                   |     file2.o/stack:WBP  |
                   +------------------------+
                   |  Ccommon:WB            |
                   +------------------------+
                   |  bss:WBC               |
                   |     file1.o/bss:WBC    |
                   |     file2.o/bss:WBC    |
                   *------------------------*
```

Figure 5-4. Selecting Sections by File

Note, however, that the combined BSS section and the Ccommon section still come at the end of this link. This is because the BSS sections are combined automatically and the Ccommon section is generated automatically; automatically-generated sections do not have a file of origin. Also, note that the "stack" section is selected along with the rest of file2.o's sections, which may not be desirable.

You can make sure that the stack is postponed until the end of the link in one of two ways: not selecting it by combining filename and attribute selection, or un-select it with the -u argument. Commands using these two techniques are shown below:

```
  mlink -i file1.o file2.o -s file1.o, file2.o,:-P -o ex3
  mlink -i file1.o file2.o -s file1.o, file2.o, -u stack -o ex3
```

5.4.5. Separate -i Arguments

In order to circumvent these effects, if desired, you can select the sections from "file1.o" before you input "file2.o". Note that in this case you can use -s with no arguments to select everything in the input list except "postpone" sections. The second -s selects all of file2.o's sections except "stack", which has the "P" ("postpone") attribute. Also note the use of -d to define a separate Ccommon section for file1's common symbols.

```
mlink -i file1.o -s -d -i file2.o -s -o ex4 -v

MAP:   ISecn OSecn  Location    Size      IFile/Name:Atts

  Output file ex4:
             0     0 L=00000000  S=00000242  file1.o/code:ANSX
             1     1 L=00000242  S=00000231  file1.o/rom:ANRS
             2     2 L=00000474  S=00000232  file1.o/data:ANSW
             3     3 L=000006a6  S=00000200  file1.o/bss:ABCNW
             -     4 L=000008a6  S=00000014  /Ccommon:ABNSW
             0     5 L=000008ba  S=00000242  file2.o/code:ANSX
             1     6 L=00000afc  S=00000234  file2.o/data:ANSW
             2     7 L=00000d30  S=00000200  file2.o/bss:ABCNW
             -     8 L=00000f30  S=00000014  /Ccommon:ABNSW
             3     9 L=00000f44  S=00002000  file2.o/stack:ABPSW

  Input files:
             0     0 L=00000000  S=00000242  file1.o/code:ANSX
             1     1 L=00000242  S=00000231  file1.o/rom:ANRS
             2     2 L=00000474  S=00000232  file1.o/data:ANSW
             3     3 L=000006a6  S=00000200  file1.o/bss:ABCNW
             0     5 L=000008ba  S=00000242  file2.o/code:ANSX
             1     6 L=00000afc  S=00000234  file2.o/data:ANSW
             2     7 L=00000d30  S=00000200  file2.o/bss:ABCNW
             3     9 L=00000f44  S=00002000  file2.o/stack:ABPSW
```

```
file  ex4:
     *-----------------------*
     |    file1.o,code:X     |
     +-----------------------+
     |    file1.o,rom:R      |
     +-----------------------+
     |    file1.o,data:W     |
     +-----------------------+
     |    file1.o,bss:WBC    |
     +-----------------------+
     | Ccommon:WB            |
     +-----------------------+
     |    file2.o,code:X     |
     +-----------------------+
     |    file2.o,data:W     |
     +-----------------------+
     | Ccommon:WB            |
     +-----------------------+
     |    file2.o,bss:WBC    |
     +-----------------------+
     |    file2.o,stack:WBP  |
     *-----------------------*
```

Figure 5-5. Separate -i Arguments

5.4.6. Selecting Sections by Attribute

It is frequently more convenient to select sections by their attributes than by their names or files of origin. In other cases it may be *necessary*, as when preparing a general-purpose command procedure in which the names of the input files might not be known. (For example, the linker's default selection is done by attribute.)

Section attributes can be combined for selection in several different ways. You may want to select all sections that have a given set of attributes, all sections that do *not* have a particular attribute, or all sections that have *either* of two or more attributes or combinations of attributes. Loosely speaking, "not" is represented by preceding an attribute by a "-" sign, and "either" (logical "or") is represented by separating two groups of attributes by a "+" sign. The attributes themselves are represented by uppercase or lowercase letters (See Section A.2 for specific information about the attributes and their meanings). Attributes in a selection are preceded by a colon (":"), which also separates them from the section name, if any.

For example, you might want to put writable but non-BSS data first, followed by read-only data, then code, then BSS and stack. The command to do this is shown below. Note that the BSS and stack are selected at the end by default, and so need not be mentioned explicitly. Note that if you did not care what order the "rom" and "code" sections came in, you could have replaced ":R :X" with either ":R+X" or ":-W".

```
mlink -i file1.o file2.o -s :W-B :R :X -o ex5 -v

MAP:   ISecn CSecn  Location   Size     IFile/Name:Atts

  Output file ex5:
           2     0 L=00000000 S=00000232 file1.o/data:ANSW
           1     1 L=00000232 S=00000234 file2.o/data:ANSW
           1     2 L=00000466 S=00000231 file1.o/rom:ANRS
           0     3 L=00000698 S=00000242 file1.o/code:ANSX
           0     4 L=000008da S=00000242 file2.o/code:ANSX
           -     5 L=00000b1c S=00000032 /Ccommon:ABNSW
           -     6 L=00000b4e S=00000400 /bss:ABCNW
           3     7 L=00000f4e S=00002000 file2.o/stack:ABPSW

  Input files:
           0     3 L=00000698 S=00000242 file1.o/code:ANSX
           1     2 L=00000466 S=00000231 file1.o/rom:ANRS
           2     0 L=00000000 S=00000232 file1.o/data:ANSW
           3     6 L=00000b4e S=00000200 file1.o/bss:ABCNW
           0     4 L=000008da S=00000242 file2.o/code:ANSX
           1     1 L=00000232 S=00000234 file2.o/data:ANSW
           2     6 L=00000d4e S=00000200 file2.o/bss:ABCNW
           3     7 L=00000f4e S=00002000 file2.o/stack:ABPSW
```

```
          file  ex5:
                *-----------------------*
                |     file1.o/data:W    |
                +-----------------------+
                |     file2.o/data:W    |
                +-----------------------+
                |     file1.o/rom:R     |
                +-----------------------+
                |     file1.o/code:X    |
                +-----------------------+
                |     file2.o/code:X    |
                +-----------------------+
                |   Ccommon:WB          |
                +-----------------------+
                |   bss:WBC             |
                +-----------------------+
                |     file2.o/stack:WBP |
                *-----------------------*
```

Figure 5-6. Selecting Sections by Attribute

5.4.7. Locating Sections at Specific Addresses

A common problem that occurs in cross-software development is when the target system has both PROM and RAM, and it is necessary to put the program in PROM and the data in RAM. The **-address** option specifies the base address of the next section to be selected, so it is used in conjunction with selection to control the addresses of sections.

Another thing most users want to do is to locate the stack as high in memory as possible; this can be done with the **-t address** option to specify the *top* address of the *last* section to be selected. The example below shows both of these section-locating techniques. (Note that we are grouping some "writable" data with "read-only" data and code in what is presumably the PROM area; this is a common technique in languages like C which allow no distinction between writable and read-only data. In such cases, tables and so on that need to be in PROM are grouped into a single file, such as file1.o in this example.)

(In the example below, the command follows the UNIX* convention in which a backslash (\) character is used to continue a long command on another line.)

```
mlink -i file1.o file2.o -0 -s :X :R file1.o/data \
      -4000 -s :W-P -d -s :P -t 0FFFF -o ex6 -v

MAP:    ISecn OSecn  Location   Size     IFile/Name:Atts

  Output file ex6:
            0     0 L=00000000 S=00000242 file1.o/code:ANSX
            0     1 L=00000242 S=00000242 file2.o/code:ANSX
            1     2 L=00000484 S=00000231 file1.o/rom:ANRS
            2     3 L=000006b6 S=00000232 file1.o/data:ANSW
            -     4 L=00004000 S=00000400 /bss:ABCNW
            1     5 L=00004400 S=00000234 file2.o/data:ANSW
            -     6 L=00004634 S=00000032 /Ccommon:ABNSW
            3     7 L=0000dffe S=00002000 file2.o/stack:ABPSW

  Input files:
            0     0 L=00000000 S=00000242 file1.o/code:ANSX
            1     2 L=00000484 S=00000231 file1.o/rom:ANRS
            2     3 L=000006b6 S=00000232 file1.o/data:ANSW
            3     4 L=00004000 S=00000200 file1.o/bss:ABCNW
            0     1 L=00000242 S=00000242 file2.o/code:ANSX
            1     5 L=00004400 S=00000234 file2.o/data:ANSW
            2     4 L=00004200 S=00000200 file2.o/bss:ABCNW
            3     7 L=0000dffe S=00002000 file2.o/stack:ABPSW
```

```
          file  ex6:
       0000      *-----------------------*
                 |     file1.o/code:X    |
                 +-----------------------+
                 |     file2.o/code:X    |
                 +-----------------------+
                 |     file1.o/rom:R     |
                 +-----------------------+
                 |     file1.o/data:W    |
                 +-----------------------+
                 -                       -
       4000      +-----------------------+
                 |  bss:WBC              |
                 +-----------------------+
                 |     file2.o/data:W    |
                 +-----------------------+
                 |  Ccommon:WB           |
                 +-----------------------+
                 -                       -
       DFFE      +-----------------------+
                 |     file2.o/stack:WBP |
       FFFF      *-----------------------*
```

Figure 5-7. Locating Sections at Specific Addresses

5.4.8. Naming and Combining Sections

It is usually not necessary to combine or rename sections in order to affect their location or order, but naming and combining can be useful if the output of the linker is a relocatable file which is going to be used as input to a subsequent link. For example, you may want to construct a library module containing the code and data from several sub-modules. In this case it may be desirable to have only a single combined code section, a single data section, and so on. An example of this is shown below. Note the use of -r to keep the resulting output file relocatable. Note that we are specifying attributes as well as section names, and that neither the names nor the attributes of the combined sections have to be the same as those of the input sections.

```
mlink -i file1.o file2.o -r -s code -n code:X \
     -s data -n data:w -s rom -n rom:R -o ex7 -v

MAP:    ISecn OSecn  Location    Size      IFile/Name:Atts

  Output file ex7:
             -     0 R=00000000 S=00000484 /code:NSX
             -     1 R=00000000 S=00000466 /data:NSW
             -     2 R=00000000 S=00000231 /rom:NRS
             -     3 R=00000000 S=00000032 /Ccommon:BNSW
             -     4 R=00000000 S=00000400 /bss:BCNW
             3     5 R=00000000 S=00002000 file2.o/stack:BPSW

  Input files:
             0     0 R=00000000 S=00000242 file1.o/code:NSX
             1     2 R=00000000 S=00000231 file1.o/rom:NRS
             2     1 R=00000000 S=00000232 file1.o/data:NSW
             3     4 R=00000000 S=00000200 file1.o/bss:BCNW
             0     0 R=00000242 S=00000242 file2.o/code:NSX
             1     1 R=00000232 S=00000234 file2.o/data:NSW
             2     4 R=00000200 S=00000200 file2.o/bss:BCNW
             3     5 R=00000000 S=00002000 file2.o/stack:BPSW


             file  ex7:
                  *-----------------------*
                  |  code:X               |
                  |    file1.o/code:X     |
                  |    file2.o/code:X     |
                  +-----------------------+
                  |  rom:R                |
                  |    file1.o/rom:R      |
                  +-----------------------+
                  |  data:W               |
                  |    file1.o/data:W     |
                  |    file2.o/data:W     |
                  +-----------------------+
                  |  Ccommon:WB           |
                  +-----------------------+
                  |  bss:WBC              |
                  |    file1.o/bss:WBC    |
                  |    file2.o/bss:WBC    |
                  +-----------------------+
                  |    file2.o/stack:WBP  |
                  *-----------------------*
```

Figure 5-8. Naming and Combining Sections

5.4.9. Overlays

In small systems it is sometimes necessary to break programs up into pieces that "overlay" or load on top of one another. A clever loading program that understands about sections could select the sections belonging to overlays out of an object file containing the whole program, but more often it is necessary to put overlays in a separate file. This can be done in a single linking step by specifying multiple output files.

A related problem is making sure that the sections that are supposed to overlay one another start at the same address. This can be done easily if we want to specify the address exactly, but more often the overlaid sections are located relative to other sections, whose size we don't care to keep track of. The **-m** (mark) option is useful here.

The techniques used for making overlays are shown below. We assume that the code and data in file2.o are needed only part of the time, and can overlay file1.o's data section, which we therefore locate *after* the Ccommon and BSS sections. We locate the stack at the high end of memory using the **-t** option. It doesn't matter which overlay file the stack goes with because, being a BSS section, no data is actually loaded into it.

```
mlink -i file1.o file2.o -s file1.o/code :R -s :B-P -d \
      -m1 -s file1.o/data -o ex8 \
      -m1 -s :X :W-B :P -t 0ffff -o ex8a -v

MAP:    ISecn OSecn  Location    Size      IFile/Name:Atts

  Output file ex8:
            0     0 L=00000000 S=00000242 file1.o/code:ANSX
            1     1 L=00000242 S=00000231 file1.o/rom:ANRS
            -     2 L=00000474 S=00000400 /bss:ABCNW
            -     3 L=00000874 S=00000032 /Ccommon:ABNSW
            2     4 L=000008a6 S=00000232 file1.o/data:ANSW

  Output file ex8a:
            0     5 L=000008a6 S=00000242 file2.o/code:ANSX
            1     6 L=00000ae8 S=00000234 file2.o/data:ANSW
            3     7 L=0000dffe S=00002000 file2.o/stack:ABPSW

  Input files:
            0     0 L=00000000 S=00000242 file1.o/code:ANSX
            1     1 L=00000242 S=00000231 file1.o/rom:ANRS
            2     4 L=000008a6 S=00000232 file1.o/data:ANSW
            3     2 L=00000474 S=00000200 file1.o/bss:ABCNW
            0     5 L=000008a6 S=00000242 file2.o/code:ANSX
            1     6 L=00000ae8 S=00000234 file2.o/data:ANSW
            2     2 L=00000674 S=00000200 file2.o/bss:ABCNW
            3     7 L=0000dffe S=00002000 file2.o/stack:ABPSW
```

```
   file  ex8:
0000     *-----------------------*
         |    file1.o,code:X     |
         +-----------------------+
         |    file1.o,rom:R      |
         +-----------------------+
         |  bss:W3C              |
         +-----------------------+
         |  Ccommon:WB           |
08A6     +-----------------------+
         |    file1.o,data:W     |
         *-----------------------*

   file  ex8a:
08A6     *-----------------------*
         |    file2.o,code:X     |
         +-----------------------+
         |    file2.o,data:W     |
         +-----------------------+
         -                       -
DFFE     +-----------------------+
         |    file2.o,stack:WBP  |
FFFF     *-----------------------*
```

Figure 5-9. Overlays

5.4.10. Discarding Sections

It is sometimes useful to produce an object file containing only some of the sections of the input files. This is an alternative way of producing overlays; it is used more often if one input file contains an operating system and another an application that runs under it. The application will need to know the addresses of routines in the operating system, but can assume that the operating system will already be in memory.

Sections are discarded by giving a -o option with no filename. This is shown in the example below, in which all the sections in file1.o are discarded.

```
mlink -i file1.o -s -d -o -i file2.o -s -o ex9 -v

MAP:    ISecn OSecn  Location    Size      IFile/Name:Atts

  Discarded:
              0     0 L=00000000 S=00000242 file1.o/code:ANSX
              1     1 L=00000242 S=00000231 file1.o/rom:ANRS
              2     2 L=00000474 S=00000232 file1.o/data:ANSW
              3     3 L=000006a6 S=00000200 file1.o/bss:ABCNW
              -     4 L=000008a6 S=00000014 /Ccommon:ABNSW

  Output file ex9:
              0     5 L=000008ba S=00000242 file2.o/code:ANSX
              1     6 L=00000afc S=00000234 file2.o/data:ANSW
              2     7 L=00000d30 S=00000200 file2.o/bss:ABCNW
              -     8 L=00000f30 S=00000014 /Ccommon:ABNSW
              3     9 L=00000f44 S=00002000 file2.o/stack:ABPSW

  Input files:
              0     0 L=00000000 S=00000242 file1.o/code:ANSX
              1     1 L=00000242 S=00000231 file1.o/rom:ANRS
              2     2 L=00000474 S=00000232 file1.o/data:ANSW
              3     3 L=000006a6 S=00000200 file1.o/bss:ABCNW
              0     5 L=000008ba S=00000242 file2.o/code:ANSX
              1     6 L=00000afc S=00000234 file2.o/data:ANSW
              2     7 L=00000d30 S=00000200 file2.o/bss:ABCNW
              3     9 L=00000f44 S=00002000 file2.o/stack:ABPSW
```

```
              file  ex9:
                    *-----------------------*
                    |    file2.o/code:X     |
                    +-----------------------+
                    |    file2.o/data:W     |
                    +-----------------------+
                    |    file2.o/bss:WBC    |
                    +-----------------------+
                    | Ccommon:WB            |
                    +-----------------------+
                    |    file2.o/stack:WBP  |
                    *-----------------------*
```

Figure 5-10. Discarding Sections

Chapter 6
MLIST

6.1. INTRODUCTION

The *mlist* utility uses special comments that the assembler can optionally insert into an object file (with the **-oson** assembler option) to construct an assembler-like listing file from a MUFOM object module.

6.2. COMMAND SYNTAX AND OPTIONS

The command syntax for this utility is as follows:

```
   mlist [-o file] [-s | -l | -x] [file]
```

The file and options may appear in any order. If no file is given, standard input is used.

The command-line options are:

-o file
  output file name (If not specified, output is to standard output.)

-s short format (***'s instead of expressions)

-l long format (single long line for overflow of object code)

-x exclude object code that doesn't fit on the source line.

6.3. USAGE, OUTPUT FORMAT AND EXAMPLES

The input file should be generated by running the assembler with the **-os -on** options, to get source code and line numbers into the object file. Most object-file utilities can be made to preserve comments with the **-k** option; the comments used by *mlist* are in levels 2 and 3, so the **-k3** option should be used. In particular, keeping comments through *mlink* means that an assembler-like listing can be generated from a fully linked and relocated load module.

A full explanation of the MUFOM variables used in the expressions displayed in the object-code column of the listing can be found in Appendix A. The more common expressions are:

```
        Xnnn                external
        Rnnn+offset         relocatable in section nnn
```

Apart from addition, represented by an infix "+" sign, operations in expressions are listed as

```
        operation(operand,operand...)
```

even for operations such as "*" for multiplication. The entire expression is enclosed in an additional set of parentheses.

The following examples show a short assembly-language program with its assembler listing, and *mlist*-generated listings in the various available formats.

```
asm80k -oson -oc foo.s -o foo.o -p

asm80k version 2.1a
Mon Apr 28 09:41:34 1986         foo
LOC       OBJ               LINE# --- SOURCE ---
                                1          .extern xxx
00000000 6121803cWwwwwwww       2          ld      r1, rr2[foo1][r3]
00000008 1402********           3 foo1:    ldl     rr2, #xxx + foo1
0000000e                        4          .blkb   1000h
0000100e R000+00000008,**       5          .dd     foo1, xxx * 100, foo1 ^<
00001013 **************
                                6
```

```
mlist foo.o

mlist v. 2.1 -- Zilog MUFOM listing utility

00000000                        1          .extern xxx
00000000 6121803c(R0+8)         2          ld      r1, rr2[foo1][r3]
00000008 1402(X0+R0+8)          3 foo1:    ldl     rr2, #xxx + foo1
0000000e                        4          .blkb   1000h
0000100e (R0+8)(*(X0,64))       5          .dd     foo1, xxx * 100, foo1 ^<
00001016 (&INS(0,R0+8,3,
00001016 4f))
```

```
mlist foo.o -s

mlist v. 2.1 -- Zilog MUFOM listing utility

00000000                        1          .extern xxx
00000000 6121803c*********      2          ld       r1, rr2[foo1][r3]
00000008 1402*********          3 foo1:    ldl      rr2, #xxx + foo1
0000000e                        4          .blkb    1000h
0000100e *****************      5          .dd      foo1, xxx * 100, foo1 ^
00001016 ********
```

```
mlist foo.o -x

mlist v. 2.1 -- Zilog MUFOM listing utility

00000000                        1          .extern xxx
00000000 6121803c(R0+8)         2          ld       r1, rr2[foo1][r3]
00000008 1402(X0+R0+8)          3 foo1:    ldl      rr2, #xxx + foo1
0000000e                        4          .blkb    1000h
0000100e (R0+8)(*(X0,c4))       5          .dd      foo1, xxx * 100, foo1 ^
```

```
mlist foo.o -l

mlist v. 2.1 -- Zilog MUFOM listing utility

00000000                        1          .extern xxx
00000000 6121803c(R0+8)         2          ld       r1, rr2[foo1][r3]
00000008 1402(X0+R0+8)          3 foo1:    ldl      rr2, #xxx + foo1
0000000e                        4          .blkb    1000h
0000100e (R0+8)(*(X0,64))       5          .dd      foo1, xxx * 100, foo1 ^
00001016 (@INS(0,R0+8,3,4f))
```

Chapter 7
MLOAD

7.1. INTRODUCTION

The *mload* utility is a format conversion program which translates MUFCM files into one of three formats suitable for moving an object module from a host system to a target system. The three output formats are Tektronix and Intel Hex formats, and a simplified version of MUFOM. *mload* is usually used in conjunction with *protocol*, which sends the resulting outout to a target system using the Tektronix or other handshaking protocol.

In addition to simply converting formats, *mload* has several options which are useful in burning PROMs and in downloading.

7.2. COMMAND SYNTAX AND OPTIONS

The command syntax for this utility is as follows:

```
        mload [options] [file]
```

If no filename is given, the standard input will be converted.

The command-line options are:

-o file
    Output file name (if not specified, output is to standard output).

-a   MUFOM absolute download subset (default)

-i   This option specifies the output to be in Intel Hex format, as defined in Appendix C.

-t   This option specifies the output to be in Tektronix Hex format, as defined in Appendix B.

(The following are useful for burning PROMs.)

-N   Output every Nth byte. Divide input addresses by *N* to get output addresses.

@H Start at (input) address H.

=H Output H bytes.

(The following options are useful for downloading, and especially for mapping code into a specific segment.)

+H Add offset of H to every *output* address.

-p (PROM) Subtract *start* from every *input* address (before the division specified by -N option). This starts output addresses at zero for burning a PROM.

-z Map Z8001-type segmented addresses into 24-bit linear addresses. The 7-bit segment number in bits 24-30 of the input address is placed in bits 16-22 of the output address. Thus, the Z8000 address 12001234 (<<12h>>1234h in assembler notation) is mapped into the output address 121234.

(The following apply only to MUFOM or Intel download formats.)

-g Output global symbols.

-l Output local symbols.

-k Keep comments of level N or lower (MUFOM only). (default: N = 255)

-s Output section information (MUFOM only).

7.3. OPERATION

7.3.1. mload Input

The input to *mload* is a single MUFOM format object module. If the input object module is relocatable (*i.e.*, there are symbols for address references for which no values have been associated), then *mload* will produce an error message but will proceed with the translation, relocating every section starting at zero.

7.3.2. mload Address Translation

The parameters that affect *mload*'s address translation are:

S the specified starting address (@S option).

L The number of bytes to be output (=L option).

T the specified offset (+T option). The -p option sets T = -S.

N the number of separate PROMS being burned (-N option).

Given an input address A, this will be translated to an output address A/N - T. Cnly data with addresses between S and S + N*L will be loaded.

7.3.3. Output Format Limitations

It should be noted that all symbolic information is lost when MUFOM is translated into Intel or Tektronix Hex. In addition, MUFOM sections have no counterpart in Intel or Tektronix format, i.e., all sections in the MUFOM file will become one contiguous set of records when translated.

Intel Hex format limits addresses to 16 bits without extended addressing, and 20 bits with extended addressing. Tektronix Hex format limits addresses to 16 bits. Thus, large programs may have to be downloaded in several pieces.

7.4. USING mload: SOME EXAMPLES

The following examples show how *mload* works. The first few examples assume the following input module called "tload.o":

```
MBZ80K,05tload.
AD08,04,M.
DT19860505094554.
C00100,15--- Section Table ---.
ST00,A,03abs.
SA00,02.
ASS00,0111.
ASL00,00.
ST01,A,X,04code.
SA01,02.
ASS01,2F002006,2F002000,-.
ASL01,2F002000.
C00101,14--- Symbol Table ---.
ASG,2F002000.
NN01,03foo.
ASN01,0101.
NI00,05start.
ASI00,2F002000.
C00102,18--- Program Sections ---.
SB00.
LR000102030405060708090A0B0C0D0E0F.
ASP00,0101.
LR0102030405060708090A0B0C0D0E0F10.
SB01.
LR5E03AF002000.
ME.
```

7.4.1. MUFOM Download Formats

The following three examples show the use of mload to produce absolute MUFOM output in a form suitable for downloading.

Command:

```
    mload tload.o -o load.o
```

Output: load.o (absolute MUFOM)

```
    MBZ80K/05tload.
    AD08/04/M.
    ASP00/00.
    LR000102030405060708090A0B0C0D0E0F.
    ASP00/0101.
    LR0102030405060708090A0B0C0D0E0F10.
    ASP00/2F002000.
    LR5E08AF002000.
    ASG/2F002000.
    ME.
```

Command:

```
    mload tload.o -o load.o -s
```

Output: load.o (MUFOM with sections)

```
    MBZ80K/05tload.
    AD08/04/M.
    C00100/15--- Section Table ---.
    ST00/A/03abs.
    SA00/02.
    ASS00/0111.
    ASL00/00.
    ST01/A/A/04code.
    SA01/02.
    ASS01/06.
    ASL01/2F002000.
    C00101/14--- Symbol Table ---.
    C00102/18--- Program Sections ---.
    SB00.
    ASP00/00.
    LR000102030405060708090A0B0C0D0E0F.
    ASP00/0101.
    LR0102030405060708090A0B0C0D0E0F10.
    SB01.
    ASP00/2F002000.
    LR5E08AF002000.
    ASG/2F002000.
    ME.
```

Command:

```
    mload tload.o -o load.o -slg
```

Output: load.o (MUFOM with sections and symbols)

```
    MBZ80K,05tload.
    AD08,04,M.
    C00100,15--- Section Table ---.
    ST00,A,03abs.
    SA00,02.
    ASS00,0111.
    ASL00,00.
    ST01,A,A,04code.
    SA01,02.
    ASS01,06.
    ASL01,2F002000.
    C00101,14--- Symbol Table ---.
    NN01,03foo.
    ASN01,0101.
    NI00,05start.
    ASI00,2F002000.
    C00102,18--- Program Sections ---.
    SB00.
    ASP00,00.
    LR000102030405060708090A0B0C0D0E0F.
    ASP00,0101.
    LR0102030405060708090A0B0C0D0E0F10.
    SB01.
    ASP00,2F002000.
    LR5E08AF002000.
    ASG,2F002000.
    ME.
```

7.4.2. Translating from MUFOM to Intel Hex

Suppose that you want to translate an object module that is formatted in MUFOM into Intel Hex records. The following example shows how this would be performed:

Command:

```
    mload -i tload.o -o load.o
```

Output: load.o (Intel Hex)

```
    mload v. 2.1 -- Zilog MUFOM load formatting utility
    :10000000000102030405060708090A0B0C0D0E0F78
    :100101000102030405060708090A0B0C0D0E0F1066
    :062000005E08AF002000A5
    :00200003DD
    :00000001FF
```

The **-i** option specifies that the input be translated into Intel hex records. **link.o** is the input file. **-o load.o** specifies that output goes into the file called load.o.

Note that the addresses in the output have been truncated to 16 bits.

7.4.3. Translating from MUFOM to Tektronix Hex

The method shown for translating object modules from MUFOM format to Intel Hex in the previous section is the same method used for translating Tektronix format. Instead of the **-i** option (output = Intel), the **-t** option is used to indicate that the output will be Tektronix format.

The following example shows the translation of a file to Tektronix Hex with output on Standard Output.

Command:

```
    mload -t tload.o
```

Output (Standard Output)

```
    mload v. 2.1 -- Zilog MUFOM load formatting utility
    /00001001000102030405060708090A0B0C0D0E0F78
    /010110030102030405060708090A0B0C0D0E0F1079
    /200006085E08AF00200036
    /20000002
```

Note that the addresses in the output have been truncated to 16 bits.

7.4.4. Downloading to a PROM Programmer

Downloading a program or segment thereof to a PROM programmer is straightforward. First, generate a file of the proper format, i.e., Intel or Tektronix Hex. Second, attach the programmer to your terminal's auxiliary port. Third, *cat* (UNIX*) or *type* (DOS) the file while capturing the data on the programmer. Last, burn the PROM. This method has been used successfully with Data I/O Programmers and ADM 31 terminals.

A second method can be used if the PROM programmer is attached to a second serial port. In this case, the output of *mload* can be sent to this port instead of to a file. If the PROM programmer requires a handshake, *protocol* can be used (see Section 10.3.3 in the chapter on *protocol*.)

7.4.5. Programming Multiple PROMs

When a program is too big to fit into a single PROM, it is necessary to perform several loads. The following example shows how to do this.

Suppose you have a file, "file1" which is to be translated into two Tektronix-format files "prom1" and "prom2" with starting addresses 0000 and 1000 (hex) respectively. You can do this with the two commands

```
        mload -p file1 -o prom1
        mload -p file1 -o prom2 @1000
```

The **@1000** option in the second command specifies that output starts with address 1000 (hex). The **-p** option specifies that the physical addresses in the output files start with 0000.

7.4.6. Programming PROMs for a 16-bit Processor

When developing software for 16-bit processors such as the Z8000, it is necessary to program odd and even locations into separate PROMs. The following example shows how to do this:

Given a file "file1" which you want to separate into two Intel-format files, "prom0" and "prom1" respectively, you use the two commands:

```
        mload -2 -p file1 -o prom0
        mload -2 -p file1 -o prom1 @1
```

The **-2** option specifies that two PROMs are being programmed, so that only every other byte is to be loaded. The **-p** option specifies that addresses in the PROM output file start with 0. The **@1** option in the second command specifies that output to file "prom1" starts with address 1 in the input file.

Note that for 32-bit processors, **-4** can be used to produce four PROMs.

7.4.7. Translating Logical to Physical Addresses

When developing software for systems that incorporate memory mapping, it is sometimes necessary to load software at a different address (physical address) from the address at which it is intended to run (logical address). The following example shows how to perform this translation using the +*offset* option:

Given a file "file1" containing a program linked starting at logical location 0, you want to load the program into physical segment 1 on a Z8001. The Z8001 CPU places the start of segment 1 at 01000000(hex); the target system's memory places it at 010000(hex). Use the command:

```
        mload file1 -z +01000000
```

The output of this command is another MUFOM file on standard output. The **-z** option specifies that 32-bit Z8001 logical addresses are mapped into 24-bit physical addresses by "squeezing out" the second byte. The **+01000000** option specifies that 01000000 is added to logical load addresses in the input file (*before* the translation implied by the **-z** option.

Note that only the addresses at which data are to be *loaded* are mapped. Addresses in the program, and the values of symbols, are unchanged.

Chapter 8
MLORDER

8.1. INTRODUCTION

The *mlorder* utility takes a list of MUFOM modules and computes the optimum order for putting these modules into a library. "Optimum order" is the order that allows all required modules to be found in a single pass through the library; thus, all modules in the library that reference a symbol appear in front of the module that defines it.

It is not always possible to find such an optimum order; *mlorder* will inform you if this is the case, with the message:

```
        cycle in data:
```

followed by a list of the modules that contain a circular series of references.

The output file generated by *mlorder* is in a form that can be used by *mlib* to generate a library.

8.2. COMMAND SYNTAX AND OPTIONS

The *mlorder* conversion utility is invoked by the following command:

```
        mlorder [-r] [file]...
```

The command-line options are:

-r   If the -r option is given, the standard output is a list of pairs of object file names, meaning that the first file of the pair refers to external identifiers defined in the second. The output may be processed by *tsort* to find an ordering suitable for one-pass access by *mlink*.

     Alternatively, the proper ordering may be generated directly by *mlorder* by not giving the -r option. In this case the output is a file suitable for direct input to *mlib* with the f option.

Chapter 9
MNM

9.1. INTRODUCTION

The *mnm* utility prints the symbol table name list for a given file in any of several formats.

9.2. COMMAND SYNTAX AND OPTIONS

The command syntax for this utility is as follows:

```
mnm [options] [file]
```

If no file is given, standard input is used.

The command-line options are:

-l   Include local symbols in the listing.

-n   List symbols in numerical order.

-u   List symbols unsorted, that is, in the order they appear in the object file.

-m   List symbols with link map information.

-s   Swapped format, with name first on the line.

-s N
     Swapped format with name first and truncated to N characters.

-o file
     Direct output to the given file instead of standard output.

9.3. OUTPUT FORMAT AND EXAMPLES

mnm displays the symbols defined in the given file in any of several formats. Options are provided to display

- only global and external symbols (the default) or local symbols as well.
- symbols in alphabetical order, in numerical order by address, or in their order of definition.
- with or without link map information.
- in a short form suitable for use with symbolic debuggers, emulators, or other utilities.

9.3.1. Default Name List Format

The default format of the symbol name list is shown in the example below. The list has a line entry for each symbol. The first column shows the value of the symbol. This is a hexadecimal number for absolute symbols, or an expression involving an R-variable (relocatable section origin) or X-variable (external symbol). More complex expressions are listed as "<expression.>".

The second column contains "X" for external symbols, "I" for global (internal) symbols, and "N" for local symbols.

The third column contains the name of the symbol.

```
mnm -l foo.o

mnm v. 2.1 -- Zilog MUFOM namelist utility
00001002+X0000          N expr1
<Expression.>           N expr2
0000000C+X0000          X ext1
00000000+X0001          X ext2
00000004+R0000          I glb1
00001002                I glb2
0000123456789abcdef     I glb3
00000005+R0000          N loc1
0000000a+R0000          N loc2
```

Note that the above example was prepared with the "-l" option to list local symbols.

9.3.2. Name List with Map Information

The **-m** option can be used to list symbols with information derived from the section table, and from the link map in modules output by **mlink**. The fourth column contains the file of origin for the symbol, with a library name in parentheses if the symbol came from a library. The fifth column contains the name of the section in which the symbol resides, and its attributes. This column contains "?:" if the section cannot be determined. (The space between columns has been decreased a little in the example below to make it fit within the margins in this manual.)

```
mnm -m foo.o

mnm v. 2.1 -- Zilog MUFOM namelist utility
00005714        I __align        doprtz.o(foo.lib), libcode:ANS
00005256        I __doprtz       doprtz.o(foo.lib), libcode:ANS
00005086        I __iob          strlen.o(foo.lib), libcode:ANS
00005786        I __prtint       doprtz.o(foo.lib), libcode:ANS
000058b2        I __xputc        doprtz.o(foo.lib), libcode:ANS
00005000        I _atoi          atoi.o(foo.lib), libcode:ANSX
00005200        I _printf        printz.o(foo.lib), libcode:ANS
0000000a        I _putc          foo.o, allfoo:ANSW
000058fa        I _strlen        strlen.o(foo.lib), libcode:ANS
00005026        I _strncmp       strncmp.o(foo.lib), libcode:AN
00000000        I foo1           foo.o, allfoo:ANSW
00000002        I foo2           foo.o, allfoo:ANSW
00000004        I foo3           foo.o, allfoo:ANSW
000067ab        I gru            strlen.o(foo.lib), ?:
00004567        I zoo            strlen.o(foo.lib), ?:
00005026        I zorch          strlen.o(foo.lib), libcode:ANS
00001234        I zork           strlen.o(foo.lib), ?:
00001234        I zorn           strlen.o(foo.lib), ?:
00005678        I zot            strlen.o(foo.lib), libcode:ANS
```

9.3.3. Swapped Name List Format

In order to interface to some symbolic debuggers, it is possible to get a "swapped" listing with the name first on the line. It is also possible to truncate the name field to a given number of characters. This is done with the **-s** or **-s N** option, as in the example below.

```
        mnm -s8 foo.o

        mnm v. 2.1 -- Zilog MUFOM namelist utility
        ext1    X 00000000+X0000
        ext2    X 00000000+X0001
        glb1    I 00000004+R0000
        glb2    I 00001002
        glb3    I 0000C123456789abcdef
```

Chapter 10
PROTOCOL

10.1. INTRODUCTION

The _protocol_ utility is the upload/download communication handshake program. It supports a variety of different file-transfer protocols commonly used on PROM programmers and development systems. It is normally used in conjunction with _mload_ to download modules into a target system.

10.2. COMMAND SYNTAX AND OPTIONS

The command syntax for this utility is as follows:

```
   protocol [options] [file]
```

A maximum of one file may be specified; if no file is specified the standard input is used for downloading, standard output for uploading. Order of command line arguments is not significant.

The command-line options are:

-d device
: download device name. (If no -d option is given or no device is specified, the terminal is used.)

-u [device]
: upload device name. (If no device is specified, the terminal is used.)

-f file
: take command arguments from the specified file. Arguments in the file may be separated by whitespace or newlines; comments start with a semicolon and end with newline.

-e
: suppress error messages.

-s string
: setup string sent to upload/download device. Multiple -s options are permitted; the strings are concatenated.

-p protocol
specifies protocol. (Default Tektronix.)

The protocol is matched with a list of protocol names. Case is ignored, and abbreviation is allowed. Presently, the only protocol defined is "Tektronix".

Protocol may also be a list of items of the form "variable=value". Values are numeric; hex if they start with "0", decimal otherwise. Variables are one of the following:

ack
acknowledgement character.

nak
negative acknowledgement character.

abort
abort character.

linedelay
delay (in milliseconds) after sending each line.

chardelay
delay (in milliseconds) after sending each character.

prompt
prompt character.

retry
number of times to retry an incorrectly-received record.

timeout
timeout in seconds.

10.3. USING protocol: SOME EXAMPLES

10.3.1. Downloading to a Z8 or Z8000 Development Module

To download an object module to a target system such as a Zilog Z8 or Z8000 development module, the following procedure is used:

(1) In Unix, create an alias with the command

```
alias LOAD 'protocol -t'
```

In other operating systems, create a command file with the same effect. Note that the filename argument to the LOAD command is appended after the "-t" option. If you want to specify MUFOM object modules rather than Tektronix hex, your alias or command file will need to run them through *mload* first; this can be done with

```
alias LOAD 'mload -t * | protocol -t'
```

(On operating systems other than Unix, this will take two commands, with *mload* creating a temporary intermediate file.)

(2) While running in the development module's monitor, execute the command:

```
LOAD <filename>
```

The development module sends the host the command:

```
LOAD <filename>
```

which the host interprets as

```
protocol -t filename
```

which performs the Tektronix handshake protocol with the development module.

10.3.2. Uploading from a Z8 or Z8000 Development Module

The procedure to upload from the Z8 or Z8000 development module is slightly more complicated than the procedure used to download. The user must know the starting and ending addresses of the image to be uploaded before proceding. Given that, the following procedure must be followed:

(1) Alias "SEND" to "protocol -u -t".

(2) While running in the development module's monitor, execute the command:

```
SEND <filename> <start-addr> <end-addr>
```

The development module sends the host the command:

```
LOAD <filename>
```

which the host interprets as

```
protocol -u -t filename
```

This invokes *protocol*, which performs the Tektronix handshake protocol with the development module. The resulting file is in Tektronix Hex format, suitable for downloading again.

NOTE that "protocol -t -u filename" is incorrect: this causes *protocol* to interpret the given filename as the device to upload from, with odd results.

10.3.3. Downloading to a PROM Programmer

Some PROM programmers do not require a download protocol; they simply have a file copied directly to them, as described in the chapter on *mload*. Others (e.g., the DATA/IO model 21) require more elaborate treatment as described below.

It is most convenient, if a device requires a complex download protocol, to make a command file. For downloading to a DATA/IO model 29 attached to device "/dev/tty4", this file (call it "dataio") should contain:

```
-d /dev/tty4
-s \E\E86A\rI\r
-p prompt=03E
```

In order to download a file, for example "foo", use the command

```
protocol -f dataio foo
```

For uploading from the DATA/IO, the corresponding command file should contain:

```
-s \E\E86A\r2000;\r1CM\r0\r
-u /dev/tty4
-p prompt=03E
```

Naturally, other PROM programmers and emulators will have different protocols; you will need to consult your manual for details, and will probably have to experiment as well.

Chapter 11
OTHER PROGRAMS

The following programs are supplied with the Object File Utilities for specialized purposes:

```
mar
m2a
muimage.c
```

They are described below.

11.1. MAR

The *mar* utility is an older version of *mlib*. It produces a so-called "archive" file which is compatible with older versions of *mlink*, as well as the library files of the Berkeley version of the UNIX* operating system. Archive files have the advantage of being able to contain any kind of file (not just MUFOM object files), and the disadvantage of not allowing the linker to access them randomly.

The command line of *mar* is identical to that of *mlib* (see Chapter 4).

11.2. M2A

The *m2a* utility converts MUFOM object files to a form called *a.out*, which is the format used in Zilog's S8000 microcomputers. This format is primarily useful for downloading into Zilog's EMS-8000 emulator for the Z8000 microprocessor.

11.2.1. Command Syntax And Options

The *m2a* conversion utility is invoked by the following command:

```
m2a [ -i | -o ] [ -s seg ] inputfile outputfile
```

The command-line options are:

-i   Put instructions and data in separate address spaces.

-o   Convert an overlay file.

-s H is the segment number (in hexadecimal) in which the stack is to reside.

The input file must be absolute, *i.e.* the output of *mlink* or *mload*. Many features of MUFOM cannot be converted to *a.out*, these include arbitrary expressions involving relocatable or external symbols, and sections other than code, data, and BSS.

11.3. MUIMAGE.C

The *muimage.c* program is the C-language source for a program. It converts a MUFOM character form object file on Standard Input to an absolute binary image file on its Standard Output, while producing a hexadecimal listing on Standard Error (the terminal). This program is not very useful by itself, but is supplied in source form so that you can construct a customized loader for whatever target system you are using. *muimage* is designed to work on the output of *mload*, and understands only absolute modules in character form.

11.3.1. Command Syntax

The *muimage* conversion program is invoked by the following command:

```
muimage [inputfile] > outputfile
```

If no input file is specified, Standard Input is used.

Appendix A
MUFOM FILE FORMAT

A.1. THE MUFOM STANDARD

The MUFOM format, as implemented by the Zilog cross-software products, follows the format specified in the IEEE standard *IEEE 695-1985*, "The Microprocessor Universal Format for Object Modules." The standard specifies only the *character form* for object files; the *binary form* of MUFOM files follows the suggested format in Appendix B of the standard.

Section A.2 discusses the concepts of modules and sections, and the various section attributes and their meanings. Section A.3 discusses the way MUFOM handles symbols, and the use of MUFOM variables. Section A.4 discusses the local usage of IEEE Standard 695 by the Zilog cross-software, including implementation restrictions. Section A.5 discusses local extensions to the standard that have been added to implement efficient library search. Section A.6 contains an example of a MUFOM object module and an explanation of its constituent commands.

A.2. MODULES AND SECTIONS

MUFOM object modules (object files) are divided into *sections* each of which is destined to be loaded into a separate area of memory. Each section has a *name*, a *size*, *attributes*, and (if not relocatable) a *location*. Each section also has a *section number* which is used to refer to it internally. In Zilog's implementation these section numbers correspond to the order of the sections in the section table. Section numbers are limited to 16 bits. The name and attributes of a section are specified in a MUFOM "ST" (Section Type) command; the size and location are MUFOM S- and L-variables respectively.

It is important to note that the name of a section may be null (in which case the section is referred to as "unnamed"), and that the names of sections *need not be unique*. Thus, a file may contain several sections named "code". The advantage of this is that the linker can relocate such sections separately. Therefore, on a Z8001 all "code" sections do not have to be in the same segment.

Sections may also have an *alignment* and *page size*. The location (lower bound) of a section is restricted to be a multiple of its alignment, and the section may not cross a boundary which is a multiple of its page size. The page size is used to implement address-space and segment-size limits. The alignment and page size of a section are specified by the MUFOM "SA" (Section Alignment) command.

The following is a description of the various section attributes and their meanings. This includes the way they affect the link process, and their eventual use in a target system. Each attribute is represented by a letter (lowercase or uppercase).

A.2.1. Access Attributes.

The access attributes specify how sections are used (accessed) in the target system. They are used during the link process to select groups of sections that are to be located together.

W (Writeable)
This is the default access attribute.

R (Read-only)
This attribute is used for data that is intended to go into ROM.

B (BSS)
This attribute is used for data that is initialized to zero when a program is started. (BSS stands for "Block Started by Symbol".)

X (Executable)
This attribute is used for code sections.

Z (Zero page)
This attribute is used for sections that are accessed via a processor-dependent short addressing mode, such as the Z8 on-chip registers.

A (Absolute)
Sections with this attribute have been located at an absolute address.

A.2.2. Overlap Attributes.

The overlap attributes specify how sections with the same name and same access attributes are to be handled. Sections can be unnamed; all unnamed sections are treated as if they have the same name. The overlap attributes are mutually exclusive and a section may have only one of them.

S (Separate)
All sections with this attribute will be kept separate when located in the output file. This is the default overlap attribute.

C (Concatenate)
Concatenate (combine into a single contiguous chunk) all sections with the same name and attributes. This attribute performs the equivalent of the linker's -n command line option.

E (Equal Length)
Overlap all sections with the same name and attributes; the size of the resulting section is the size of its components. Produce an error message if they have different sizes.

M (Maximum Length)
Overlap all sections with the same name and attributes; the size of the resulting section is the size of its largest component.

U (Unique Names)
Only one section with the same name and attributes is permitted.

A.2.3. Allocation Attributes.

The two allocation attributes determine the order in which sections are selected.

N (Now)
Selected sections with the "n" attribute will be merged before all sections with the "p" attribute. This is the default allocation attribute.

P (Postpone)
Selected sections with the "p" attribute will be merged after all sections with the "n" attribute. When sections are selected via mlink's -s command-line argument, any "postpone" sections selected are placed after any "now" sections selected by the same sub-argument. Thus,

```
-s code data
```

selects first the sections with name "code" and attributes that include "n", then sections with name "code" and attribute "p", then sections with name "data" and attribute "n", and finally sections with name "data" and attribute "p".

A.3. SYMBOLS AND VARIABLES

MUFOM modules associate numerical values with constructs called *variables*, which are represented as a letter in the set G-Z followed by a hexadecimal number, the *index*. In Zilog's representation, variable indices are restricted to 16 bits. (Avoiding the letters A-F as variable identifiers means that variables can always be distinguished from hexadecimal numbers.)

Values are assigned to variables with the MUFOM "AS" (Assign) command.

A.3.1. Section Variables

Some of the variables in a MUFOM module are associated with sections, and their index is the same as the corresponding section number. These variables, and their meanings, are:

S Size of the corresponding section.

L Location of the corresponding section. The L-variable is present only for absolute sections.

R "Relocation base" for the section. In absolute sections this is initialized to the section's location; in relocatable sections it represents the address at which the section will eventually be located.

P "Program Counter" for the section. In the load data of the module, the P-variable represents the next location at which code will be loaded. Space can be reserved within a section by assigning to the P-variable.

A.3.2. Symbol Variables

Other variables in a MUFOM module are associated with symbols. The value assigned to the variable is the value of the corresponding assembly-language symbol.

The symbol variables are:

N N-variables are associated with *local* symbols (names). It is possible to have more than one N-variable in a module with the same name; this occurs when two modules containing local symbols with the same name are linked together.

I   I-variables are associated with *global* (Internal) symbols. These are symbols defined within a module that can be referred to in other modules that are linked with it.

X   X-variables are associated with *external* references to global symbols in other modules. X-variables are never assigned values.

As implemented at Zilog, the N, I, and X variables of any object module are allocated contiguously starting from N0, I0, and X0. The variable indices do *not*, however, necessarily correspond to the order of the variables in the symbol table. It is only guaranteed that there will be no gaps in the numbering.

A.3.3. Other Variables

Finally, there are two other kinds of variables in MUFOM modules:

G   There is at most one G-variable in a MUFOM module; its value is the *entry point* or starting address of the program.

W   W-variables are "working registers". W0 is used as temporary storage for range-checking. The other W-variables are used by the assembler to hold the values of *forward references*, that is, symbols that are used before they are defined.

A.4. LOCAL USAGE

A.4.1. Comments

There are two special local usages for comments. Comment levels 0-3 are used for specific kinds of debugging and link map information. Comment levels 100(hex)-102 are used for separating the object file into regions containing different kinds of information.

A.4.1.1. Information Comments

Information comments contain error messages, source code, and link map information. They allow symbolic debuggers and other programs (including *mdump*, *mm*, and *mlist*) to display more information than would otherwise be present in the object file.

0 Comment level 0 is used for error messages.

1 Comment level 1 contains comments of the following forms:

:FILE Innnn Nnnnn Dnnnnnnnnnnnn filename(library)
Input file information. I, N, and D precede the I-variable origin, N-variable origin, and creation date respectively.

:SECT isecn osecn L=nnnnnnnn S=nnnnnnnn ifile,secname:atts
Link map information on input sections.

filename: line-number
Compiler filename and line number from .FILE and .LINE assembler statements.

2 Comment level 2 contains assembler source lines.

3 Comment level 3 contains assembler listing format information, in comments of the form:

filename: linenumber
Assembler source file and line number.

:PAGE
marks a new listing page.

A.4.1.2. Object File Regions

Three special comments divide the object file into regions, as shown in Figure A-1. The region before the first such comment is the file header, containing the MB, AD, and DT commands.

100 A comment of level 100(hex) introduces the section table, containing ST, SA, ASL, and ASS commands and the input file and link map comments.

101 A comment of level 101(hex) introduces the symbol table, containing NI, NX, NN, ASI, ASN, ASG, WX, AT, LI, LX, RI, and TY commands.

102 A comment of level 102(hex) introduces the load data region, containing ASW, SB, ASP, LD, LR, IR, and RE commands and comments containing error messages, assembler source, and so on.

Splitting object files into these regions makes the linker and other utilities more efficient by marking parts of the file that do not need to be processed.

```
--------------------------------
|                              |
|        Module Header         |
|                              |
--------------------------------

--------------------------------
|                              |
|        Section Table         |
|                              |
--------------------------------

--------------------------------
|                              |
|        Symbol Table          |
|                              |
--------------------------------

--------------------------------
|                              |
|       Program Section        |
|                              |
--------------------------------

--------------------------------
|          Module End          |
--------------------------------
```

Figure A-1. Cbject Module Regions

A.4.2. Expressions

The MUFOM standard permits the use of expressions of great generality in many places in the object files. What the linker and other utilities will *accept* is, in general, more restricted; and what the assembler, linker, and other utilities will *emit* is more restricted still.

In this section, codes are used to indicate the kinds of expressions that are acceptable. Except when only hexnumbers are permitted, *all* functions are allowed. The codes are:

Hnn	any hexnumber of at most nn bits.
R	R-variables.
W	W-variables.
X	X-variables.

As a rule, other variables are not used in expressions; all utilities expand them into equivalent expressions involving

R, W, and X variables. W00 is used purely as a temporary for limit checking; other W variables are used by the assemblers for forward references.

Indices	All variable indices and section numbers are H16's.
Addresses	All addresses are limited to 32 bits (H32 RWX).
Numbers	Numbers up to 80 bits (H80) are permitted in assignments and LR commands.
SA	Section Alignment: H32, Page Size: H32.
CO	Comment level: H16.
AS(L,S)	H32.
AS(G,I,N)	H80 RX.
AS(R,P)	H32 R.
ASW	H80 RWX.
LR	H80 RWX.
IR	Relocation Base: H80 RWX, Number Of Bits: H8.
RE	H32.
WX	H80 RWX.

A.4.3. Command Order

There are some local restrictions on the ordering of MUFOM commands. Observing these restrictions makes it possible to avoid retaining information that will not be needed later.

A.4.3.1. Section Information

All the information for a single section is grouped together, with the ST command first, followed by the SA, ASL and ASS commands in any order. The ASS command must be present, and the section size must be a hexnumber.

A.4.3.2. Variable Information

All the information for a single variable is grouped together, with the NV command first.

An NI, or NN command is always followed immediately by the corresponding AT command (if any) and ASI or ASN command.

An NX command is always followed immediately by the corresponding AT and WX commands, if any.

W-variables must be assigned to before they are referenced.

R-variables are assigned implicitly. No existing utilities generate ASR commands.

A.5. LIBRARY EXTENSIONS

The following commands have been added for maintaining libraries. A library file consists of a library header, the modules in the library, and a library map. The library header consists of an LB command, an optional DT command, and an LE command. The library map is at the end of the file, and consists of a series of LM commands followed by an LE command. The library extension commands are always in character form. Modules contained in a library may either be in character form or binary form.

LB map_position "," lib_name "."

Library Begin: the first command in a library. The hexnumber map_position is the position in the file of the library map, which consists of LM commands followed by an LE command.

The LB command may be followed by a DT command with the date the library was created or last modified. This is followed by an LE, the MUFOM modules contained in the library, and the library map.

LE "."

Library End: marks the end of the library header and the library map.

LM position "," size "," m_name ("," I_name)* ("," "X" ("," X_name)*) "."

Library Module: indicates the position in the library of a module, its size in bytes, its name, the names of the symbols (I-variables) it defines, and the names of the external symbols (X-variables) it references.

A.6. EXAMPLE: Zilog MUFOM Module

Figure A-2 shows an actual character-form MUFOM module produced by a Zilog MUFOM Cross-assembler. Line numbers have been added in parentheses along the right margin for reference purposes.

```
  MBZ800,06link.o.                                          (1)
  DT19850522102255.                                         (2)
  AD08,02,L.                                                (3)
  C00100,17---- Section Table ----.                         (4)
  ST00,X,A,04code.                                          (5)
  ASL00,00.                                                 (6)
  ASS00,31.                                                 (7)
  ST01,W,A,04data.                                          (8)
  ASL01,31.                                                 (9)
  ASS01,0C.                                                (10)
  ST02,B,W,C,A,06COMMON.                                   (11)
  ASL02,3D.                                                (12)
  ASS02,10.                                                (13)
  C00101,16---- Symbol Table ----.                         (14)
  NN01,08bc_store.                                         (15)
  ASN01,31.                                                (16)
  NN02,05bc$hl.                                            (17)
  ASN02,28.                                                (18)
  NI00,03div.                                              (19)
  ASI00,00.                                                (20)
  NI01,09dvd_store.                                        (21)
  ASI01,33.                                                (22)
  NN03,04div1.                                             (23)
  ASN03,09.                                                (24)
  NN04,04div2.                                             (25)
  ASN04,1B.                                                (26)
  NI02,09mpy_store.                                        (27)
  ASI02,30.                                                (28)
  ATI02,00,00,10,02.                                       (29)
  ASG,00.                                                  (30)
  C00102,1A---- Program Sections ----.                     (31)
  SB00.                                                    (32)
  ASP00,00.                                                (33)
  LREBC5444D2100003E10B7CB13CB12CB15CB14CD2800DA1800       (34)
  ED421C3DC20900C1EB223300223D00C9E5B7ED42223100E1C9.      (35)
  SB01.                                                    (36)
  ASP01,31.                                                (37)
  LR000000000000000000000000.                              (38)
  SB02.                                                    (39)
  ASP02,3D.                                                (40)
  C0FF,20BSS (uninitialized data) section.                 (41)
  ME.                                                      (42)
```

Figure A-2. MUFOM Module

Lines (1) - (3) define the module header. The module header is standard across all Zilog MUFOM object modules.

MB - Module Begin

- o Defines the start of a MUFOM object module.
- o Defines the target processor type (optional).
- o Defines the object module's name (optional).

DT - Date

- Defines the object module's creation time and date.

AD - Address Descriptor

- Defines the number of bits per Minimum Addressable Unit (MAU).
- Defines the maximum size of the target processor's address space in MAUs (optional).
- Defines the order of the address's MAUs within the object code.

Line (4) - MUFOM comment commands are used in the Zilog MUFOM implementation to delimit the different parts of the object module. Line (4) delimits the start of the Section Table. Comments are prefixed by the MUFOM CO command.

Lines (5) - (13) are the section table. Each section within an object module will have a set of description commands in this table. MUFOM commands that are used in the section table region of the module are

ST - Section Type. For a given section, defines

- Section type attributes (optional)
- Section name (optional)

SA - Section Alignment. For a given section, defines

- Section alignment (given in MAUs) (optional)
- Maximum section size (optional)

AS - Assignment

- Assigns values to section variables
  S - section size
  L - location of section's lower boundary (optional)

Line (14) - The start of the symbol table.

Lines (15) - (30) - The symbol table contains declarations for all of the global, external, and local symbols within the object module. Assignment of absolute values or expressions to symbols, definition of symbol type, and definition of symbol attributes are kept here. The module's entry point, if any, is also specified here. MUFOM commands used in the symbol table region of the object file are

NI - Name Internal (Global) Symbol

- Declares an external symbol, a table entry number, a name length count, and gives its name.

NX - Name External Symbol

- Declares an external symbol, a table entry number, a name length count, and gives its name.

NN - Name Local Symbol

- Declares a local symbol, a table entry number, a name length count, and gives its name.

AT - Symbol Attribute. Defines for a given symbol
- o Type table entry
- o Lexical level (optional)
- o Size (used for common symbols) (optional)
- o Alignment (used for processors where variables must be aligned on specific addresses) (optional)

AS - Assignment.
- o Assigns a value or expression to a symbol.

TY - Type. Defines a type table entry.

WX - Weak External.
- o Defines a given symbol as a weak external*.
- o Provides a default value or expression to be assigned if the symbol is not resolved.

Line (31) The MUFOM comment used to delimit the start of the program portion of the object module.

Lines (32) - (41) The code, or load data, portion of the object module is kept in this region. The heading "Program Sections" refers to the MUFOM sections which are the logical divisions of the program. MUFOM commands used within this region to define the code are

SB - Section Begin
- o Declares that the following code belongs to the specified section.

LD - Load.
- o Contains object code.

LR - Load Relocate.
- o Contains code and relocation expressions.

IR - Initialize Relocation Base
- o Assigns a value to a relocation letter.

RE - Replicate.
- o Repeat the immediately following LR expression a specified number of times.

AS - Assignment
- o Assigns a value or an expression to a section's P (load pointer) variable.

---

* A weak external will be resolved with a global definition if one is present; otherwise it receives the default value specified in the WX command.

Line (42) The end of a MUFOM object Module is delimited by the ME Command.

Appendix B
TEKTRONIX HEX FORMAT

B.1. RECORD FORMAT

Record Format

field name:	SR	ADDR	RL	CS1	****DATA****	CS2	END
field size:	1	4	2	2	0-255	2	1

Figure B-1. Tektronix Hex Record Format

SR--Start Record Field (frame 0)
The ASCII slash character (/) is used to signal the start of a record.

ADDR--Load Address Field (frames 1 to 4)
The starting location in memory to/from which data will be loaded/saved.

RL--Record Length Field (frames 5 and 6)
Tha number of data bytes in the record is represented by two ASCII hexadecimal digits.

CS1--First Checksum Field (frames 7 and 8)
This checksum is the 8-bit sum of the six hexadecimal digits that make up the load address and record length.

DATA--Data Field (frames 9 to 9 + (RL * 2) -1)
Each pair of frames in the data field represents a data byte, where each frame contains the ASCII representation of a 4-bit value.

CS2--Second Checksum Field (frames (9 + (RL * 2)) and (9 + (RL * 2)) + 1)
This checksum is the sum of the 4-bit hexadecimal values of the digits in the data field, modulo 256.

END--End of Record Field (frame (9 + (RL * 2)) + 2)
The ASCII code for a carriage return is used to signal the end of a record.

B.2. END-OF-FILE RECORD FORMAT

The end-of-file record has a record length of 0, the address field containing the entry point address, and no data or second checksum.

Example B-1: /4F000013<CR>

B.3. ABORT RECORD FORMAT

The download operation can be aborted by the host system sending an abort record, consisting of two slashes followed by an error message and carriage return.

Example B-2: //PROGRAM ABORTED <CR>

B.4. HANDSHAKING FOR DOWNLOAD/UPLOAD

mload and msend use the Tektronix handshaking protocol, by default, for each format. Since there is no handshaking used in conjunction with the Intel Hex format, the -h option must be used to turn it off whenever the -i option is specified. The handshaking protocol consists of three signals:

o "0" No error o "7" Bad record; retransmit o "9" Abort

These signals are sent by the target to the host when downloading and vice versa when uploading.

It is recommended that handshaking always be used to prevent erroneous data transmission.

Appendix C
INTEL HEX FORMAT

C.1. RECORD FORMAT

Record Format

field name:	SR	RL	ADDR	RT	****DATA****	CS
field size:	1	2	4	2	0-255	2

Figure C-1. Intel Hex Record Format

SR--Start Record Field (frame 0)
The ASCII character colon (:) is used to signal the start of a record.

RL--Record Length Field (frames 1 and 2)
The number of data bytes in the record is represented by two ASCII hexadecimal digits.

ADDR--Load Address Field (frames 3 to 6)
Four ASCII hexadecimal digits representing zeros or the address to/from which data will be loaded/saved.

RT--Record Type Field (frames 7 and 8)
The ASCII hexadecimal digits in this field specify one of the record types shown in Table C-1:

Table C-1. Intel Hex Record Types

Record Type	Description
00	Data Record
01	End-of-File Record
02	Extended Address Record
03	Start Address Record (entry point)

The address specified by the Extanded Address Record is left-shifted four bits (representing the four most significant bits in a 20-bit address), and added to all subsequent type 00 (Data Record) addresses.

DATA--Data Field

Each pair of frames in the data field represents a data byte, where each frame contains the ASCII representation of a 4-bit value.

CS--Checksum Field

This field contains the ASCII representation of the two's complement of the sum of the data bytes (each pair of data field frames converted to one binary byte), modulo 256.

Appendix D
ERROR MESSAGES

D.1. INTRODUCTION

Each utility describes errors with clearly stated error messages. There are three types of errors that can occur: process errors, input format errors, and internal errors.

The action taken due to an error depends on the severity of the error and the utility being executed. Most errors do not interrupt object module processing.

D.1.1. Process Errors

Process errors occur due to either incorrect command usage, or otherwise-correct command usage on inappropriate data (for example, attempting to load a relocatable file).

Process errors can occur

- While attempting to interpret the command line used to invoke the utility.
- During the processing of object modules.

D.2. COMMON ERRORS

The following errors are common to most or all of the utilities:

D.2.1. Command Line Errors

The common command line errors are

```
-<letter> argument filename missing
-<letter> argument number missing
garbage after numeric argument: -<letter>
unrecognized command-line argument -<letter>
-<letter> argument inconsistent with previous arguments
extra output filename ignored
```

D.2.2. Other Errors

```
OPEN error on file <file name>
can't handle libraries
division by zero
no free storage left
value out of range
```

An OPEN error means either that a specified input file does not exist or is protected against reading, or that a specified output file is protected against writing.

"No free storage left" means that there are too many symbols, sections, or files in the input.

The "division by zero" and "value out of range" errors represent errors in assembly-language code which could not be detected by the assembler because they involved relocatable or external symbols.

D.3. COMMAND-SPECIFIC ERRORS

D.3.1. mlib Errors

The errors unique to mlib are

```
unknown option '<letter>'
Can not read '<filename>'
Must have exactly one of 'd,q,r,t,x'
No archive file specified
Only one option allowed in 'd,q,r,t,x'
archive file '<filename>' not found
missing argument for 'f' options
multiple '<letter>' options
BuildLM - '<filename>' Not archive format
BuildLM - Out of memory
BuildLM - no matching LE
CreatMlib - Can not create '<filename>'
Out of memory
WriteAll - Can not create '<filename>'
WriteAll - Can not open '<filename>'
can't handle libraries
q_mlib - Out of memory
x_mlib - Can not create '<filename>'
```

D.3.2. mlink Errors

The errors unique to *mlink* are

```
- without attribute in select string <arg>
-m not implemented in relocatable link
-t argument address missing
-<symbol> =value with no symbol
-<symbol> length with no symbol
-t cannot relocate absolute section
-t with no sections selected
+symbol: symbol missing
=value: value missing
E section attribute: sections must have same size
U section attribute: sections must be unique
attempting to merge
    absolute section    <section descriptor>
    with reloc. section <section descriptor>
entry point <symbol> undefined
file <filename> has different address order
illegal character in select string <arg>
multiply-defined global symbol <symbol>
nested -f files not allowed
output file <filename> is also an input file
symbol <symbol> not absolute
undefined external <symbol>
```

D.3.3. mlink Warnings

The only utility that generates warnings, as opposed to errors, is *mlink*. Warnings represent unusual conditions that may, nevertheless, be what you intended to produce. The **-w** option to *mlink* suppresses these warning messages.

```
address space overflow at <address>
attempting to load into BSS section at <address>
no section information for section <number>
no section information in input file <filename>
null select: <arg>
null unselect: <arg>
section overlap
    <section descriptor>
    <section descriptor>
symbol <symbol> redefined by -g argument
```

The "section overlap" error, in particular, can occur when making files with separate address spaces for instruction and data. The "no section information" errors occur when linking files generated as output from *mload* without the **-s** option.

D.3.4. mlist Errors

There are no errors actually unique to *mlist*, but errors included as level-0 comments by the assembler are included in the listing.

D.3.5. mload Errors

The errors unique to *mload* are

```
section <name> is relocatable
```

D.3.6. mlorder Errors

The errors unique to *mlorder* are

```
<symbol> multiply defined in <filename> and <filename>
cannot open <filename>
cycle in data:
extern overflow
module overflow
out of memory
symbol space overflow
text space overflow
```

D.3.7. protocol Errors

The errors unique to *protocol* are

```
too many files
write record error
write first '0' error
cannot open <filename>
conflict -d & -u options
duplicate -<letter> options
invalid -r option - <string>
invalid -t option - <string>
no file specified
unknown handshaking code <number> from Remote
unrecognized option <string>
```

D.4. INPUT FILE FORMAT ERRORS

Input File Format Errors are primarily associated with the parsing and execution of MUFOM commands (as opposed to utility program command lines) within a MUFOM object file. These errors are displayed in one of two formats:

```
    input file format error:  MCE at line 9 of foo.o
```

or

```
    input file format error:  MCE at line 9 of foo.o
         MCE: missing command-end period
         byte 0xa of the MUFOM command:
    NNC1,xxx,?04foo1.
```

The second, more descriptive format is obtainable via the -v4 option in *mlink*. If the input file is in binary form, the line number is replaced by an offset in characters from the beginning of the file.

It is generally impossible to get format errors unless a MUFOM file has been garbled, or generated incorrectly. This usually is caused by a bug in one of the utilities, and should be called to the attention of your Zilog representative.

Table D-1. Input File Format Errors.

2HD:	2 hex digits required
ADR:	address > 32 bits
ARG:	not enough arguments for function
ASG:	multiple assignments to G-variables (entry points)
ASI:	variable index of ASI does not match previous NI
ASL:	ASL command before or without ST
ASS:	ASS command before or without ST
ASX:	assignment to X-variable is illegal
CMD:	command expected/undefined command
EOF:	unexpected end-of-file in <filename>
EXP:	expressions not permitted in download
EXU:	expression stack underflow
IAF:	invalid archive format
IAH:	invalid archive header
ILF:	invalid library format
LIB:	library command inside module
MAU:	can't handle MAU length other than 8 bits
MCE:	missing command-end period
MCP:	missing comma or period
MCS:	missing command start
MEX:	missing expression
MMB:	MB command missing
MRO:	missing relocation offset
MRP:	missing ')'
MSA:	SA command with no expressions
MST:	no ST for section <number>
N16:	number > 16 bits
N32:	number > 32 bits
N80:	number > 80 bits
NAN:	not a number
NNR:	not a number or R-variable
STL:	string > 127 characters long
TYU:	unexpected TY-component
TYV:	N-variable or T-number expected
UAT:	AT command does not apply to previous variable
UEX:	unknown/invalid item in expression
UFN:	unknown operator/function
ULD:	invalid load item
USA:	SA command before or without ST
UXP:	unexpected referance to P-variable
VAR:	undefined variable
XEX:	too many expressions
XMB:	MB command not at beginning of file
XRP:	unexpected ')'

D.5. INTERNAL ERRORS

Internal errors generally indicate a bug in one of the utilities; they represent conditions that should not occur, and should be called to the attention of your Zilog representative.

```
READ error on file <filename>
<upload/download> read error
write error
core dumped
```

The "core dumped" error is a host operating system error which usually means that something drastic is wrong with the program, but it can also occur if a program runs out of free storage and fails to detect the fact.

GLOSSARY

absolute code: Code whose position within memory has been defined and whose address references have been assigned values relative to the code's position.

absolute loader: A process which can load one or more sections of absolute code only at the locations specified by the sections.

checksum: A semi-random function of a file's contents. If a file is copied and the checksum of the copy is different from that of the original, there has been an error in copying.

code: A program or segment thereof which has been encoded in a language useable by a processor. Often used loosely as a synonym for "load data". See Object Code, Source Code.

command: Control information for a linker or loader. It is to be distinguished from Load Data.

external reference: The usage, within a module, of a symbol which is defined outside that module. An imported global definition.

file: A MUFOM object file is a structure defined by the host operating system containing one or more MUFOM object modules. Files containing more than one module are considered to be libraries.

global definition: The definition within a module, of a symbol which may be used outside that module.

identifier: A string of characters which uniquely represents a defined entity such as a symbol, option or command.

library: A set of two or more object modules.

linker: A program that combines object modules into a single object module satisfying links between the object modules.

load data: Data (including machine instructions) to be loaded into a processor's memory.

load pointer: A pointer for a section which is dynamically maintained by the loader. It indicates where the next item of the code is to be loaded. It is initialized to a starting load address.

local symbol: A symbol which is accessible only within a single module.

machine code: Code that is directly understandable by a processor's hardware. Since digital processors are binary in nature, machine code consists of binary numbers. See Object Code.

module: A program or portion thereof, usually in the form of a separate file. See Object Module, Source Module.

object code: Code (Load Data) contained in an Object Module.

object format: The language in which Object Modules are specified.

object module: A MUFOM object module is a set of sections of absolute or relocatable machine code, together with ancillary commands. See Module, Source Module.

prelink: A link session that precedes one or more other link sessions over the same object code.

program: An algorithm and associated data. A series of operations to be performed over some given data.

process: A program executed by a processor.

relocatable code: Code that consists of machine code and relocation commands. Relocation commands allow address references within the machine code to be reevaluated if the machine code is repositioned in memory. Relocatable code is to be distinguished from absolute code.

section: A part of a program with ancillary information (commands) which becomes a segment when loaded.

segment: A contiguous region in memory with arbitrary boundaries which may contain machine code.

source code: A program in some human-readable programming language. Source code is translated into Object Code by a compiler or assembler.

source module: A Module containing Source Code.

symbol: A label or name that represents a numeric value.

symbol resolution: The process of replacing an external reference with its globally defined value.

Z8® Microcontroller Technical Description I

Zilog Z8® Software II

Zilog General Information III

General Terms and Conditions of Sale

ORDERING PRODUCTS

Orders placed for Zilog components should include the component part number as shown in the example below. The part number consists of a "Z" prefix, followed by a five-digit part number, two-digit numerical speed designator, alpha package designator, alpha operating temperature range designator, and an environmental flow designator (e.g., Z8032008VSC or Z0840006VEC).

ORDERING CODES

PACKAGE

IC PACKAGE CODES

A = VQFP (Very Small QFP)
C = Ceramic Side Brazed
D = Cerdip
E = Ceramic Window
F = Plastic Quad Flat Pack
G = Ceramic PGA (Pin Grid Array)
H = SSOP (Slim Small Outline Package)
I = PCB Chip Carrier
K = Cerdip Window
L = Ceramic LCC (Leadless Chip Carrier)
P = Plastic DIP
S = SOIC (Small Outline Integrated Circuit)
V = Plastic Leaded Chip Carrier

SUPPORT TOOL PACKAGE CODES

T = Emulation Module
Z = Support Tools

ENVIRONMENTAL

PREFERRED
C = Plastic Standard
E = Hermetic Standard

LONGER LEAD TIME
A = Hermetic Stressed
B = 883 Class B Military
D = Plastic Stressed

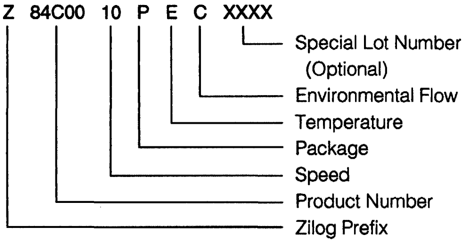
TEMPERATURE

PREFERRED
Standard: S = 0°C to +70°C

LONGER LEAD TIME
Extended: E = –40°C to +100°C
(–40°C to +105°C for Consumer Products)
Military: M = –55°C to +125°C

EXAMPLE

Z84C0010PEC is a CMOS 8400, 10 MHz, Plastic, –40°C to +100°C, Plastic Standard Flow.



III

General Terms and Conditions of Sale

1. **Terms:** Net 30 days

2. **Order/Shipment Minimums:**

A. Commercial Standard Product

- $500 per order
- $250 per line item and/or shipment release
- 100 piece minimum quantity/line item per release in multiples of tube, tray, or reel count

B. Custom ROM Products

- 10,000 unit order minimum for 18-, 28-, or 40-pin devices
- One-half of the units to be scheduled within ninety (90) days
- $3,000 mask charge for each new ROM

C. Non-Standard Product

- Windowed Products
- Systems
- Development Boards
- Emulators
- Software

100 piece minimum waived
$250 line item minimum still applies

D. Tape and Reel

44-lead PLCC	500 units per reel minimum
68-lead PLCC	250 units per reel minimum

E. Trays

- 44-lead QFP = 96 pieces per tray.
- 80-lead QFP = 50 pieces per tray.
- 100-lead QFP = 50 pieces per tray.
- 48-lead VQFP = 60 pieces per tray.
- 100-lead VQFP = 90 pieces per tray.

F. Technical Publications

- $100 per order or shipment release

3. Cancellation, Reschedule, and Failure to Release

If buyer cancels shipment of any purchase order or a portion of any purchase order or reschedules without prior agreement by Zilog, any purchase order or a portion of any purchase order, the following charges may, at Zilog's option, be assessed and invoiced by Zilog:

Product Type	*Notice Received Prior to Acknowledgment Shipping Date	Cancellation Reschedule Charges
Commercial	0 - 30 Days	No cancellations allowed. 100 per cent Invoice charges apply.
Military	0 - 90 Days	No cancellations allowed. 100 per cent Invoice charges apply.
ROM*	0 - 90 Days	No cancellations allowed. 100 per cent Invoice charges apply.
Remote Control End Products	0 - 90 Days	No cancellations allowed. 100 per cent Invoice charges apply.

Note:
* Notice shall be calculated from the customer request date.

ROM Code Variations

Because ROM Coded Products are custom products made specifically for Buyer, Buyer agrees that Zilog may ship a quantity of such ROM Coded Products which is five percent (5%) more or less than the quantity ordered and that such variation will be accepted as delivery in full and paid for by Buyer.

Zilog price quotations and acknowledgments are dependent upon quality and schedule. If the Buyer does not release the full quantity quoted and acknowledged within the time frame stated on the quotation, Seller reserves the right to either invoice the full quantity quoted and acknowledged within the time frame stated on the quotation or to invoice for a higher price in accord with Seller's price schedule for the lower quantity actually released by Buyer.

4. Product Availability

Product availability is a function of a constantly changing market and manufacturing conditions, therefore Zilog cannot guarantee availability. Please contact your local Zilog sales office or sales representative for current product availability information.

Information for products not listed in this selection guide can be obtained from your local Zilog sales office, or sales representative. The point of delivery will be determined by the Zilog sales order acknowledgment.

5. Cost Adders

Special processing of both commercial and military products to the customer's specifications (non-Zilog standard) is available in the following circumstances on most Zilog products: top mark, packing instructions, shipping instructions, one lot date code per shipment, stepping qualification, and certificate of conformance (C of C). Read Only Memory (ROM) mask charges are required for ROM coded products. For information regarding charges and possible delays which special processing may have on delivery dates, contact your local Zilog sales office or sales representative. All prices quoted apply to orders placed worldwide, excluding VAT, tax, freight, duties, and exchange rate variations.

Special Services and Prices

Military Grade Components - The following cost adders should be used if standard military specifications are not adequate for a given requirement:

Condition	Charge
Generic Data	
1. Group "A" - Sample Electrical Test, per generic part type	$100.00
2. Group "B" - Assembly Construction Test, per generic part type	$100.00
3. Group "C" - 12 week results on JAN product/Die Life Test - 52 week results on non-JAN product	$100.00
4. Group "D" - 26 week results on JAN product/Package Life Test - 52 week results on non-JAN product	$100.00
5. Generic Data Pack - Includes Groups A, B, C, D data	$300.00
Customer Specific Data	
1. Group "A" - done on customer parts	$100.00
2. Group "B" - done on customer parts	$600.00
3. Group "C" - done on customer parts (per device type). Delivery increased eight weeks.	$1200.00
4. Group "D" - done on customer production lot, excludes destructive test part cost of 50 parts at customer's price. Delivery increased three weeks.	$2500.00
Additional Requirements	
1. Particle Noise Detection (PIND) testing Minimum charge per line item, per part, per order. Lot acceptance will conform to 883 Rev. C method 2020.5 allowing up to 25% lot defective maximum, pass on 1% PDA.	$250.00 minimum or 25.00 per unit
2. X-ray screening per Mil Std 883C or 5.00 per unit	$500.00 minimum
3. Lead finish other than solder dipped	Contact Factory
4. Special top marking requirements or 2.50 per unit	$250.00 minimum

Special Services and Prices

The final character in the DESC drawing number ("X") refers to the type of lead finish the parts must have. An "X" indicates that any lead finish (Solder = "A," Tin Plate = "B," Gold Plate = "C") is acceptable. It is the standard policy of Zilog to only offer the "A" lead finish which is solder dipped (ex. 5962-8551802QA).

Notes: In general, if special processing is required and is not listed above, it is not available. However, call your local Zilog sales office to discuss requirements as necessary.

Condition	Charge
Initial customer qualification of products in place of Zilog qualification report.	Customer pays for qualification sample
Customer Change Notification	
1. Notification to customer of product tooling revision	0.10 per unit
2. Notification to customer of process change	0.10 per unit
3. Customer approval of process tooling revision	0.30 per unit
4. Customer approval of process change	0.20 per unit
Special customer top mark & special customer logo (case by case basis for some requests)	0.10 per unit
Special customer burn-in in place of Zilog standard	0.50 per unit
Special customer final test	0.50 per unit
Final test data recording	1.00 per unit
Test data recording before and after burn-in	2.00 per unit
Special shipping containers	Cost plus 15%
Special shipping container marking in place of Zilog standard	0.05 per unit
Special safety stock in place of Zilog standard	0.20 per unit
Special shipping routine to point-of-title transfer in place of Zilog standard	0.10 per unit
Date code requirement in place of Zilog standard	0.05 per unit
Certificate of Origin with shipment	20.00 per shipper
Certificate of Conformance	5.00 per shipper

Special Services and Prices

Condition	Charge
Tape and Reel (where available)	
- 44-lead PLCC 500 units per reel minimum	0.10 per unit
- 68-lead PLCC 250 units per reel minimum	0.20 per unit
Special tube stoppers - rubber plugs	0.05 per unit
Special 100% full functional final test at hot temperature before burn-in	0.05 per unit
Special die orientation - die bonded upside down and rotated 90 degrees from JEDEC standards	0.10 per unit
Special back mark instruction	0.10 per unit
Special shipping box - parts to be shipped in a box lined with conductive material or static shielding bags	0.05 per unit
"Dry Pack" of PLCCs in place of normal	0.30 per unit
Special tube orientation indicator mark	0.05 per unit
Parts requiring retest	10.00 per military unit, 0.30 per commercial unit
Programming Z8/OTP	500.00 minimum per order
Failure Analysis	200.00-600.00 for military, depending on test requirements 100.00-400.00 for commercial, depending on test requirements
Single date code per shipment/line item	500.00 minimum or 5.00 per unit

Shipping Requirements for Plastic Packaging

Trays:

A	100	VQFP:	90/tray	450/bag
	48	VQFP:	60/tray	600/bag
	64	VQFP:	160/tray	800/bag
F	100	QFP:	66/tray	660/bag
	132	QFP:	36/tray	360/bag
	144	QFP:	24/tray	240/bag
	80	QFP:	66/tray	660/bag
	44	QFP:	96/tray	960/bag
H	20	SSOP:	68/tray	

I	20	PCB	Chip Carrier (C3)	(not shipping yet):	40/rail
	28	PCB	Chip Carrier (C3)	(not shipping yet):	40/rail
	44	PCB	Chip Carrier (C3)	(Not shipping yet):	30/rail

P	18	Plastic DIP:	20	units/rail
	20	Plastic DIP:	20	units/rail
	28	Plastic DIP:	15	units/rail
	40	Plastic DIP:	10	units/rail
	48	Plastic DIP:	10	units/rail
	52	Plastic DIP:	10	units/rail
	64	Plastic DIP:	10	units/rail

S	18	SOIC	40	units/rail	1000/bag
	20	SOIC:	38	units/rail	950/bag
	28	SOIC:	27	units/rail	1080/bag
V	44	PLCC:	25	units/rail	500/bag
	68	PLCC:	20	units/rail	400/bag
	84	PLCC:	15	units/rail	225/bag

Tape and Reel:

S	18	SOIC:	2,000/reel
	20	SOIC:	2,000/reel
V	44	PLCC:	500/reel
	68	PLCC:	250/reel
	84	PLCC:	250/reel

ZILOG DOMESTIC SALES OFFICES AND TECHNICAL CENTERS

CALIFORNIA	
Agoura	818-707-2160
Campbell	408-370-8120
Irvine	714-453-9701
San Diego	619-658-0391
COLORADO	
Boulder	303-494-2905
FLORIDA	
Clearwater	813-725-8400
GEORGIA	
Duluth	404-931-4022
ILLINOIS	
Schaumburg	708-517-8080
MINNESOTA	
Minneapolis	612-944-0737
NEW HAMPSHIRE	
Nashua	603-888-8590
OHIO	
Independence	216-447-1480
OREGON	
Portland	503-274-6250
PENNSYLVANIA	
Horsham	215-784-0805
TEXAS	
Austin	512-343-8976
Dallas	214-987-9987

INTERNATIONAL SALES OFFICES

CANADA	
Toronto	905-850-2377
CHINA	
Shenzhen	86-755-2220869 86-755-2220873
Shanghai	86-21-415-0691 86-21-415-8158 Rm. 5204
GERMANY	
Munich	49-8967-2045
Sömmerda	49-3634-23906
JAPAN	
Tokyo	81-3-5272-0230
HONG KONG	
Kowloon	85-2-2723-8979
KOREA	
Seoul	82-2-577-3272
SINGAPORE	
Singapore	65-2357155
TAIWAN	
Taipei	886-2-741-3125
UNITED KINGDOM	
Maidenhead	44-628-392-00

The information in this document is subject to change without notice. Devices sold by Zilog, Inc. are covered by warranty and patent indemnification provisions appearing in Zilog, Inc. Terms and Conditions of Sale only. Zilog, Inc. makes no warranty, express, statutory, implied or by description, regarding the information set forth herein or regarding the freedom of the described devices from intellectual property infringement. Zilog, Inc. makes no warranty of merchantability or fitness for any purpose. Zilog, Inc. shall not be responsible for any errors that may appear in this document. Zilog, Inc. makes no commitment to update or keep current the information contained in this document.

Zilog's products are not authorized for use as critical components in life support devices or systems unless a specific written agreement pertaining to such intended use is executed between the customer and Zilog prior to use. Life support devices or systems are those which are intended for surgical implantation into the body, or which sustains life whose failure to perform, when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in significant injury to the user.

**Zilog, Inc. 210 East Hacienda Ave.
Campbell, CA 95008-6600
Telephone (408) 370-8000
Telex 910-338-7621
FAX 408 370-8056
Internet: http://www.zilog.com/zilog**

SALES REPRESENTATIVES AND DISTRIBUTORS

U.S., CANADIAN & PUERTO RICAN REPRESENTATIVES

ALABAMA

Huntsville

Alabama Bits, Inc	(205) 534-4020

ARIZONA

Scottsdale

Thom Luke Sales, Inc.	(602) 451-5400

CALIFORNIA

Irvine

Infinity Sales	(714) 833-0300

Santa Clara

Phase II Technical Sales	(408) 980-0414

San Diego

Addem	(619) 729-9216

COLORADO

Englewood

Thorson Rocky Mountain	(303) 773-6300

CONNECTICUT

Wallingford

Advanced Technical Sales	(508) 664-0888

FLORIDA

Altamonte Springs

Semtronic Associates, Inc.	(407) 831-8233

Clearwater

Semtronic Associates, Inc.	(813) 461-4675

Fort Lauderdale

Semtronic Associates, Inc.	(305) 731-2484

GEORGIA

Norcross

BITS	(404) 564-5599

ILLINOIS

Hoffman Estates

Victory Sales, Inc.	(708) 490-0300

IOWA

Cedar Rapids

Advanced Technical Sales	(319) 393-8280

KANSAS

Olathe

Advanced Technical Sales	(913) 782-8702

MARYLAND

Pasadena

Electronic Engineering & Sales	(410) 255-9686

MASSACHUSETTS

North Reading

Advanced Technical Sales	(508) 664-0888

MICHIGAN

Novi

Rathsburg Associates, Inc.	(810) 615-4000

MINNESOTA

Minneapolis

Professional Sales for Industry	(612) 944-8545

MISSOURI

Bridgeton

Advanced Technical Sales	(314) 291-5003

NORTH CAROLINA

Huntsville

BITS	(205) 881-2900

Raleigh

BITS	(919) 676-1880

NEW JERSEY

Cherry Hill

Tritek	(609) 667-0200

NEW MEXICO

Albuquerque

Quatra & Associates	(505) 296-6781

NEW YORK

Fairport

L-Mar Associates, Inc.	(716) 425-9100

OHIO

Independence

Rathsburg Associates, Inc.	(216) 447-8825

SALES REPRESENTATIVES AND DISTRIBUTORS

OKLAHOMA

Tulsa	
Nova Marketing, Inc.	(918) 660-5105

OREGON

Portland	
Phase II Technical Sales	(503) 643-6455

TEXAS

Austin	
Nova Marketing, Inc.	(512) 343-2321
Dallas	
Nova Marketing, Inc.	(214) 265-4630
Houston	
Nova Marketing, Inc.	(713) 240-6082

UTAH

Salt Lake City	
Thorson Rocky Mountain	(801) 264-9665

WASHINGTON

Kirkland	
Phase II Technical Sales	(206) 821-8313

WISCONSIN

Brookfield	
Victory Sales, Inc.	(414) 789-5770

CANADA

British Columbia	
J-Squared Technologies, Inc.	(604) 473-4666
Ontario	
J-Squared Technologies, Inc.	(905) 672-2030
Ottawa	
J-Squared Technologies, Inc.	(613) 592-9540
Quebec	
J-Squared Technologies, Inc.	(514) 694-8330

PUERTO RICO

Rio Piedras	
Semtronic Associates, Inc.	(809) 766-0700

SALES REPRESENTATIVES AND DISTRIBUTORS

U.S. AND CANADIAN DISTRIBUTORS

NATIONWIDE

Newark Electronics	1-800-367-3573
Zeus Electronics	1-800-524-4735

ALABAMA

Birmingham

Newark Electronics	(205) 979-7003

Huntsville

Anthem Electronics	(205) 890-0302
Arrow Electronics	(205) 837-6955
Newark Electronics	(205) 837-9091

Mobile

Newark Electronics	(205) 471-6500

ARKANSAS

Little Rock

Newark Electronics	(501) 225-8130

ARIZONA

Phoenix

Anthem Electronics	(602) 966-6600
Arrow Electronics	(602) 431-0030
Newark Electronics	(602) 864-9905

Tempe

Anthem Electronics	(602) 966-6600
Arrow Electronics	(602) 431-0030
Newark Electronics	(602) 966-6340

CALIFORNIA

Arcadia

Newark Electronics	(818) 445-1420

Calabasas

Arrow Electronics	(818) 880-9686

Chatsworth

Anthem Electronics	(818) 775-1333

Chula Vista

Newark Electronics	(619) 691-0141

Fremont

Arrow Electronics	(510) 490-9477

Garden Grove

Newark Electronics	(714) 893-4909

Hayward

Arrow Electronics	(510) 487-8416

Irvine

Anthem Electronics	(714) 768-4444
Arrow Electronics	(714) 587-0404
Zeus Electronics	(714) 581-4622

Palo Alto

Newark Electronics	(415) 812-6300

Riverside

Newark Electronics	(909) 784-1101

Sacramento

Anthem Electronics	(916) 624-9744
Newark Electronics	(916) 565-1760

San Diego

Anthem Electronics	(619) 453-9005
Arrow Electronics	(619) 565-4800
Newark Electronics	(619) 453-8211

San Jose

Anthem Electronics	(408) 453-1200
Arrow Electronics	(408) 441-9700
Zeus Electronics	(408) 629-4789

Santa Clara

Newark Electronics	(408) 988-7300

Santa Fe Springs

Newark Electronics	(310) 929-9722

Ventura

Newark Electronics	(805) 644-2265

West Hills

Newark Electronics	(818) 888-3718

COLORADO

Denver

Newark Electronics	(303) 373-4540

Englewood

Anthem Electronics	(303) 790-4500
Arrow Electronics	(303) 799-0258

CONNECTICUT

Bloomfield

Newark Electronics	(203) 243-1731

Norwalk

Zeus Electronics	(203) 852-5411

Wallingford

Arrow Electronics	(203) 265-7741

Waterbury

Anthem Electronics	(203) 575-1575

SALES REPRESENTATIVES AND DISTRIBUTORS

FLORIDA

Altamonte Springs		
	Anthem Electronics	(407) 831-0007
Clearwater		
	Anthem Electronics	(813) 538-4157
		(800) 359-3522
Fort Lauderdale		
	Anthem Electronics	(305) 484-0990
Deerfield Beach		
	Arrow Electronics	(305) 429-8200
Jacksonville		
	Newark Electronics	(904) 399-5041
Orlando		
	Newark Electronics	(407) 896-8350
Plantation		
	Newark Electronics	(305) 424-4400
Tampa		
	Newark Electronics	(813) 287-1578
Lake Mary		
	Arrow Electronics	(407) 333-9300
	Zeus Electronics	(407) 333-3055

GEORGIA

Duluth		
	Anthem Electronics	(404) 931-3900
		(800) 293-0023
	Arrow Electronics	(404) 497-1300
Norcross		
	Newark Electronics	(404) 448-1300

IDAHO

Boise		
	Newark Electronics	(208) 342-4311

ILLINOIS

Addison		
	Newark Electronics	(708) 495-7740
Arlington Heights		
	Newark Electronics	(708) 956-9270
Itasca		
	Arrow Electronics	(708) 250-0500
	Zeus Electronics	(708) 595-9730
Rockford		
	Newark Electronics	(815) 229-0225
Schaumberg		
	Anthem Electronics	(708) 884-0200
	Newark Electronics	(708) 310-8980
Springfield		
	Newark Electronics	(217) 787-9972
Willowbrook		
	Newark Electronics	(708) 789-4780
		(708) 654-8250

INDIANA

Ft. Wayne		
	Newark Electronics	(219) 484-0766
Indianopolis		
	Arrow Electronics	(317) 299-2071
	Newark Electronics	(317) 259-0085
		(317) 884-0047

IOWA

Bettendorf		
	Newark Electronics	(319) 359-3711
Cedar Rapids		
	Arrow Electronics	(319) 395-7230
	Newark Electronics	(319) 393-3800
West Des Moines		
	Newark Electronics	(515) 222-0700

KANSAS

Lenexa		
	Anthem Electronics	(913) 599-1528
	Arrow Electronics	(913) 541-9542
Overland Park		
	Newark Electronics	(913) 677-0727

KENTUCKY

Louisville		
	Newark Electronics	(502) 423-0280

LOUISIANA

Metarie		
	Newark Electronics	(504) 838-9771

MARYLAND

Columbia		
	Anthem Electronics	(410) 995-6640
	Arrow Electronics	(410) 596-7800
Hanover		
	Newark Electronics	(410) 712-6922

MASSACHUSETTS

North Reading		
	Advanced Technical Sales	(508) 664-0888
Marlborough		
	Newark Electronics	(508) 229-2200
Methuen		
	Newark Electronics	(508) 683-0913
Wilmington		
	Anthem Electronics	(508) 657-5170
	Arrow Electronics	(508) 658-0900
	Zeus Electronics	(508) 658-4776
Woburn		
	Newark Electronics	(617) 935-8350

SALES REPRESENTATIVES AND DISTRIBUTORS

U.S. AND CANADIAN DISTRIBUTORS

MICHIGAN

Grand Rapids

Newark Electronics	(616) 954-6700

Livonia

Anthem Electronics	(313) 347-4090 (800) 359-3526
Arrow Electronics	(313) 462-2290

Oak Park

Newark Electronics	(810) 967-0600 (810) 968-2950

Plymouth

Arrow Electronics	(313) 462-2290

Saginaw

Newark Electronics	(517) 799-0480

MINNESOTA

Eden Prairie

Anthem Electronics	(612) 946-4826
Arrow Electronics	(612) 941-5280

Minneapolis

Newark Electronics	(612) 331-6350

St. Paul

Newark Electronics	(612) 631-2683

MISSISSIPPI

Ridgeland

Newark Electronics	(601) 956-3834

MISSOURI

Maryland Heights

Newark Electronics	(314) 298-2505

St. Louis

Arrow Electronics	(314) 567-6888

MONTANA

Helena

Newark Electronics	(406) 443-6192

NEBRASKA

Omaha

Newark Electronics	(402) 592-2423

NEVADA

Las Vegas

Newark Electronics	(702) 597-0330

Reno

Newark Electronics	(702) 322-6090

Sparks

Arrow Electronics	(702) 331-5000

NEW HAMPSHIRE

Nashua

Newark Electronics	(603) 888-5790

NEW JERSEY

East Brunswick

Newark Electronics	(908) 937-6600

Marlton

Arrow Electronics	(609) 596-8000

Pinebrook

Anthem Electronics	(201) 227-7960
Arrow Electronics	(201) 227-7880

Union

Newark Electronics	(908) 851-2290

SALES REPRESENTATIVES AND DISTRIBUTORS

NEW MEXICO

Albuquerque	
Newark Electronics	(505) 828-1878

NEW YORK

Bohemia	
Newark Electronics	(516) 567-4200
Brookhaven	
Arrow Electronics	(516) 924-9400
Cheektowaga	
Newark Electronics	(716) 862-9700
Commack	
Anthem Electronics	(516) 864-6600
Hauppauge	
Arrow Electronics	(516) 231-1000
Latham	
Newark Electronics	(518) 783-0983
Liverpool	
Newark Electronics	(315) 457-4873
Long Island	
Anthem Electronics	(516) 864-6600
Melville	
Arrow Electronics	(516) 391-1300
Rochester	
Arrow Electronics	(716) 427-0300
Pittsford	
Newark Electronics	(716) 381-4244
Port Chester	
Zeus Electronics	(914) 937-7400
Wappingers Falls	
Newark Electronics	(914) 298-2810

NORTH CAROLINA

Charlotte	
Newark Electronics	(704) 535-5650
Greensboro	
Newark Electronics	(910) 294-2142
Raleigh	
Anthem Electronics	(919) 782-3550 (800) 359-3532
Arrow Electronics	(919) 876-3132
Newark Electronics	(919) 781-7677

OHIO

Centerville	
Arrow Electronics	(513) 435-5563
Cincinnati	
Newark Electronics	(513) 772-8181
Cleveland	
Newark Electronics	(216) 391-9330
Columbus	
Newark Electronics	(614) 326-0352
Dayton	
Newark Electronics	(513) 294-8980
Solon	
Arrow Electronics	(216) 248-3990
Toledo	
Newark Electronics	(419) 866-0404
Youngstown	
Newark Electronics	(216) 793-6134

OKLAHOMA

Oklahoma City	
Newark Electronics	(405) 843-3301
Tulsa	
Arrow Electronics	(918) 252-7537
Newark Electronics	(918) 252-5070

OREGON

Beaverton	
ALMAC/Arrow Electronics	(503) 629-8090
Anthem Electronics	(503) 643-1114
Portland	
Newark Electronics	(503) 297-1984

PENNSYLVANIA

Allentown	
Newark Electronics	(610) 434-7171
Fort Washington	
Newark Electronics	(215) 654-1434
Horsham	
Anthem Electronics	(215) 443-5150
Pittsburgh	
Arrow Electronics	(412) 856-9490
Newark Electronics	(412) 788-4790

SALES REPRESENTATIVES AND DISTRIBUTORS

U.S. AND CANADIAN DISTRIBUTORS

SOUTH CAROLINA

Greenville	Newark Electronics	(803) 288-9610

TENNESSEE

Brentwood	Newark Electronics	(615) 371-1341
Knoxville	Newark Electronics	(615) 588-6493
Memphis	Arrow Electronics	(901) 367-0540
	Newark Electronics	(901) 396-7970

TEXAS

Austin	Anthem Electronics	(512) 388-0049
	Arrow Electronics	(512) 835-4180
	Newark Electronics	(512) 338-0287
Carrollton	Arrow Electronics	(214) 380-9049
	Zeus Electronics	(214) 380-4330
Corpus Christi	Newark Electronics	(512) 857-5621
Dallas	Newark Electronics	(214) 458-2528
El Paso	Newark Electronics	(915) 772-6367
Houston	Arrow Electronics	(713) 647-6868
	Newark Electronics	(713) 894-9334
Richardson	Anthem Electronics	(214) 238-7100
San Antonio	Newark Electronics	(210) 734-7960

UTAH

Salt Lake City	Anthem Electronics	(801) 973-8555
	Arrow Electronics	(801) 973-6913
	Newark Electronics	(801) 261-5660

VIRGINIA

Herndon	Newark Electronics	(703) 707-9010
Richmond	Newark Electronics	(804) 282-5671
Roanoke	Newark Electronics	(703) 772-6821

WASHINGTON

Bellevue	ALMAC/Arrow Electronics	(206) 643-9992
	Newark Electronics	(206) 641-9800
Bothell	Anthem Electronics	(206) 483-1700
Spokane	ALMAC/Arrow Electronics	(509) 924-9500
	Newark Electronics	(509) 327-1935

WEST VIRGINIA

Charleston	Newark Electronics	(304) 345-3086

WISCONSIN

Brookfield	Arrow Electronics	(414) 792-0150
Green Bay	Newark Electronics	(414) 494-1400
Madison	Newark Electronics	(608) 221-4738
Milwaukee	Newark Electronics	(414) 453-9100

CANADA

Alberta	Future Electronics	(403) 250-5550
	Future Electronics	(403) 438-2858
British Columbia	Arrow Electronics	(604) 421-2333
	Future Electronics	(604) 294-1166
Manitoba	Future Electronics	(204) 944-1446
Montreal	Arrow Electronics	(514) 421-7411
	Future Electronics	(514) 694-7710
Ontario	Arrow Electronics	(613) 226-6903
	Arrow Electronics	(905) 670-7769
	Future Electronics	(905) 612-9200
	Future Electronics	(613) 820-8313
	Newark Electronics	(519) 685-4280
	Newark Electronics	(905) 670-2888
Toronto	Arrow Electronics	(416) 670-2010
Quebec	Arrow Electronics	(418) 871-7500
	Future Electronics	(418) 877-6666
	Newark Electronics	(514) 738-4488

SALES REPRESENTATIVES AND DISTRIBUTORS

CENTRAL AND SOUTH AMERICA

MEXICO

Semiconductores Profesionales	525-524-6123

ARGENTINA

Buenos Aires

YEL SRL	011-541-440-1532

BRAZIL

Sao Paulo

Nishicom	011-55-11-535-1755
Graftec	011-572-2727
DSD Microtechnology Distributors	305-563-8665

ASIA-PACIFIC

AUSTRALIA

R&D Electronics	61-3-558-0444
GEC Electronics Division	61-2-638-1888

CHINA

Beijing

China Electronics Appliance Corp.	86-755-335-4214
TLG Electronics, Ltd.	85-2-388-7613

HONG KONG

Electrocon Products, Ltd.	85-2-481-6022
Components Agent, Ltd.	85-2-487-8826
Maxisum, Ltd.	85-2-410-2780
MEMEC, Ltd.	85-2-410-2777

INDIA

Bangalore

Maxvale	91-80-556-6761
Zenith Technologies Pvt. Ltd.	91-80-558-6782

Bombay

Zenith Technologies Pvt. Ltd.	91-22-494-7457
Maxvale	91-22-830-0959

New Delhi

Maxvale (S) Pte. Ltd.	91-11-622-5122

INDONESIA

Jakarta

Cinergi Asiamaju	62-21-7982762

JAPAN

Tokyo

Teksel Co., Ltd.	81-3-5467-9000
Internix Incorporated	81-3-3369-1105
Kanematsu Elec. Components Corp.	81-3-3779-7811

Osaka

Teksel Co., Ltd.	81-6368-9000

KOREA

ENC-Korea	822-523-2220
MEMEC, Ltd.	822-518-8181

MALAYSIA

Kuala Lumpor	60-3-703-8498
Penang L.T. Electronics Ltd.	60-4-656-2895

NEW ZEALAND

GEC Electronics Division	64-9-526-0107

PHILIPPINES

Alexan Commercial	63-2-241-9493
Cinergi Tech & Devices (Phils), Inc.	63-2-817-9519

SINGAPORE

Cinergi Technology & Devices Pte. Ltd.	65-778-9331
Eltee Electronics Ltd.	65-283-0888
MEMEC, Ltd.	65-222-4962

TAIWAN (ROC)

Acer Sertek, Inc.	886-2-501-0055
Asec Int'l. Corporation	886-2-786-6677
MEMEC, Ltd.	886-2-760-2028
Promate Electronics Co. Ltd.	886-2-659-0303

THAILAND

Eltee Electronics Ltd.	66-2-933-7565

Sales Representatives and Distributors

EUROPE

AUSTRIA

Vienna

EBV Elektronik GMBH	43-222-8941-774
Avnet/Electronic 2000	0043-1-9112847

BELGIUM

Antwerp

D & D Electronics PVBA	32-3-8277934

Zaventem

EBV Elektronik	322-7209936

DENMARK

Brondby

Ditz Schweitzer AS	4542-453044

Lynge

Rep Delco	45-35-821200

ENGLAND

Berkshire

Future Electronics	44-753-521193
Gothic Crellon	44-734-787848
Macro Marketing	44-628-604383

Kent

Arrow Electronics	44-732-74039

Lancashire

Complementary Technologies Ltd.	44-942-274731

FINLAND

Espoo

Yleiselektroniikka	358-0-452-621

FRANCE

Cedex

A2M	331-395-49-113
CCI Electronique	331-46744700

Champs sur Marne

EBV Elektronik	331-646-88600

Massy

Reptronic SA	331-60139300

GERMANY

Berlin

EBV Elektronik GMBH	030-3421041
Avnet/Electronic 2000	030-2110761

Burgwedel

EBV Elektronik GMBH	05139-80870

Camberg

Thesys A/E	49-6434-5041

Castrop

Future GMBH	02305-42051

Dortmund

Future GMBH	02305-42051

Duesseldorf

Avnet/Electronic 2000	0211-9200385
Thesys/AE	0211-536020

Erfurt

Thesys	0361-4278100

Erkrath

Avnet/Electronic 2000	211-92003-85

Frankfurt

EBV Elektronik GMBH	069-785037
Avnet/Electronic 2000	069-9738041
Future GMBH	06121-54020
Thesys/AE	06434-5041

Gerlingen

Avnet/Electronic 2000	7156-356190

Hamburg

Avnet/Electronic 2000	040-64557021

Leonberg

EBV Elektronik GMBH	07152-30090

Muenchen

Avnet/Electronic 2000	089-4511004
EBV Elektronik GMBH	089-456100
Future GMBH	089-957270
Thesys A/E	89-99355866

Nuernberg

Avnet/Electronic 2000	0911-9951610

Neuss

EBV Elektronik GMBH	02131-96770

Quickborn

Future GMBH	4106-71022

Rauxel

Future GMBH	02305-42051

Stuttgart

Avnet/Electronic 2000	07156-356190
Future GMBH	0711-830380
Thesys/AE	0711-9889100

Weissbach

EBV Elektronik GMBH	036-426486

Sales Representatives and Distributors

ISRAEL

RDT	972-35483137

ITALY

Milano

Avnet EMG S.R.L.	0039-295-343600
EBV Elektronik	0039-2-66017111
Silver Star	02-66-125-1

Firenze

EBV Elektronik	0039-55-350792

Roma

EBV Elektronik	0039-6-2253367

Modena

EBV Elektronik	0039-59-344752

Napoli

EBV Elektronik	0039-81-2395540

Torino

EBV Elektronik	0039-11-2161531

NETHERLANDS

EBV Elektronik	313-465-2353

NORWAY

Bexab Norge	47-63833800

POLAND

Warsaw

Gamma Ltd.	004822-330853

PORTUGAL

Amadora

Amitron-Arrow.	0035-1-4714806

RUSSIA

Woronesh

Thesys/Intechna	0732553697

Vyborg

Gamma Ltd.	81278-31509

St. Petersburg

Gamma Ltd.	812-5311402

SPAIN

Barcelona

Amitron-Arrow S.A.	0034-3-4907494

Madrid

Amitron-Arrow S.A.	0034-1-3043040

SWEDEN

Bexab Sweden AB	46-8-63088-00
Rep Delco Sweden AB	46-8-63086-00

SWITZERLAND

Dietikon

EBV Elektronik GMBH	0041-1-7401090

Lausanne

EBV Elektronik AG	0041-21-3112804

Regensdorf

Eurodis AG	0041-1-8433111

UKRAINE

Kiev

Thesys/Mikropribor	44-434-9533

Z8® MICROCONTROLLERS - CONSUMER FAMILY OF PRODUCTS

Databooks By Market Niche	Part No	Unit Cost
Z8® Microcontrollers Databook	DC-8305-03	$ 5.00

Product Specifications

Z86B07 CMOS Z8 8-Bit MCU for Battery Charging and Monitoring
Z86C05/C07 CMOS Z8 8-Bit Microcontroller
Z86E07 CMOS Z8 8-Bit OTP Microcontroller
Z86C11 CMOS Z8 Microcontroller
Z86C12 CMOS Z8 In-Circuit Microcontroller Emulator
Z86C21 8K ROM Z8 CMOS Microcontroller
Z86E21 CMOS Z8 8K OTP Microcontroller
Z86C61/62/96 CMOS Z8 Microcontrollers
Z86E61/63 16K/32K EPROM CMOS Z8 Microcontrollers
Z86C63/64 32K ROM Z8 CMOS Microcontrollers
Z86C91 CMOS Z8 ROMless Microcontroller
Z86C93 CMOS Z8 Multiply/Divide Microcontroller
Z86117/717 Z8 8-Bit CMOS OTP/ROM Microcontrollers

Application Notes

On-Chip Oscillator Design
Designing a Low-Cost Thermal Printer

Support Product Specifications

Z0860000ZCO Evaluation Board
Z86C1200ZEM Emulator
Z86E0700ZDP Adaptor Kit
Z86E2100ZDF Adaptor Kit
Z86E2100ZDP Adaptor Kit
Z86E2100ZDV Adaptor Kit
Z86E2101ZDP Adaptor Kit
Z86E2101ZDV Adaptor Kit
Z86C6100TSC Emulator
Z86C6200ZEM Emulator
Z86C9300ZEM Emulator
Z8 S Series Emulators, Base Units and Pods

Additional Information

Zilog's Superintegration™ Products Guide
General Terms and Conditions of Sale
Zilog's Sales Offices, Representatives and Distributors
Literature Guide & Third Party Support Vendors

Z8® MICROCONTROLLERS - CONSUMER FAMILY OF PRODUCTS

Databooks By Market Niche	Part No	Unit Cost
Infrared Remote (IR) Controllers Databook	**DC-8301-04**	**$ 5.00**

Product Specifications

Z86L03/L06 Low Voltage CMOS Consumer Controller Processor
Z86L29 6K Infrared (IR) Remote (ZIRC™) Controller
Z86L70/L71/L72/L75/L76 Zilog IR (ZIRC™) CCP™ Controller Family
Z86L73/74/77 24/32K ROM Infrared Remote Controller (ZIRC™)
Z86E72/E73/E74/77 Zilog IR (ZIRC™) CCP™ Controller Family
Z86C72/76 Zilog Infrared Remote Controller Family (ZIRC™)
Z86L78 16K, 20-Pin Zilog Infrared Remote Controller (ZIRC™)

Application Note

Beyond the 3 Volt Limit
X-10 Compatible Infrared Remote Control

Support Product Specifications

Z86C50000ZEM Emulator
Z86L7100ZDB Emulator Board
Z86L7100ZEM ICEBOX™ In-Circuit Emulator Board

Additional Information

Zilog's Superintegration™ Products Guide
Literature Ordering Guide
Zilog's Sales Offices, Representatives and Distributors

Z8® MICROCONTROLLERS - CONSUMER FAMILY OF PRODUCTS

Databooks By Market Niche	Part No	Unit Cost
Discrete Z8® Microcontrollers	**DC 8318-02**	**$ 5.00**

Product Specifications

Z86C03/C06 CMOS Z8® 8-Bit Consumer Controller Processors
Z86E03/E06 CMOS Z8® 8-Bit OTP Consumer Controller Processors
Z86C04/C08 CMOS Z8® 8-Bit Low Cost 1K/2K ROM Microcontrollers
Z86E04/E08 CMOS Z8® 8-Bit OTP Microcontrollers
Z86C07 CMOS Z8® 8-Bit Microcontroller
Z86E07 CMOS Z8® 8-Bit OTP Microcontroller
Z86C30/C31 CMOS Z8® 8-Bit Consumer Controller Processors
Z86E30/E31 CMOS Z8® 8-Bit OTP Consumer Controller Processors
Z86C40 CMOS Z8® 4K ROM Consumer Controller Processor
Z86E40 CMOS Z8® 8-Bit OTP Consumer Controller Processor

Z8® Microcontrollers Application Notes

Timekeeping with the Z8®
Using The Zilog Z86C06 SPI Bus
DTMF Tone Generation Using the Z8® CCP™
Serial Communications Using the Z8® CCP™ Software UART
The Versatile Z86C08: Three Key Features of this Z8® MCU
The Z86C08 Controls a Scrolling LED Message Display
Interfacing LCDs to the Z8® Microcontroller

Support Product Specifications and Third-Party Vendors

Z86C0800ZCO Evaluation Board
Z86C0800ZDP Adaptor Kit
Z86C1200ZEM Emulator
Z86E0600ZDP Adaptor Kit
Z86E0700ZDP Adaptor Kit
Z86E3000ZDP Adaptor Kit
Z86E4000ZDF Adaptor Kit
Z86E4000ZDP Adaptor Kit
Z86E4000ZDV Adaptor Kit
Z86E4001ZDF Adaptor Kit
Z86E4001ZDV Adaptor Kit
Z86CCP00ZEM Emualtor
Z86CCP00ZAC Emulator Kit
Z8® S Series Emulators, Base Units and Pods
Third-Party Support Vendors

Additional Information

Zilog's Superintegration™ Products Guide
Literature Guide and Ordering Information
Zilog's Sales Offices, Representatives and Distributors

Z8® MICROCONTROLLERS - CONSUMER FAMILY OF PRODUCTS

Databooks By Market Niche	Part No	Unit Cost
Digital Television Controllers	**DC-8308-01**	**$ 5.00**

Product Specifications

Z89300 Series Digital Television Controller
Z86C27/97 CMOS Z8® Digital Signal Processor
Z86C47/E47 CMOS Z8® Digital Signal Processor
Z86127 Low Cost Digital Television Controller
Z86128/228 Line 21 Closed-Caption Controller (L21C™)
Z86227 40-Pin Low Cost (4LDTC™) Digital Television Controller

Support Product Specifications

Z86C2700ZCO Application Kit
Z86C2700ZDB Emulation Board
Z86C2702ZEM In-Circuit Emulator

Additional Information

Zilog's Superintegration™ Products Guide
Literature Guide and Ordering Information
Zilog's Sales Offices, Representatives and Distributors

Telephone Answering Device Databook	**DC-8300-03**	**$ 5.00**

Product Specifications

Z89165/166 (ROMless) Low-Cost DTAD Controller (Preliminary)
Z89167/169 Z89168 (ROMless) Enhanced Dual Processor Tapeless TAM Controller (Preliminary)

Development Guides

Z89165 Software Developer's Manual
Z89167/169 Software Developer's Manual

Technical Notes

Z89165/167/169 Design Guidelines
Z89167/169 Codec Interfacing Preliminary
Controliing the Out -5V and Codec Clock Signals for Low-Power Halt Mode
Z89165/166 Input A/D and Electronic Hybrid
Z89C67/C69/167/169 Low-Power Halt Mode Sequence
Samsung KT8554 Codec
Watch-Dog Timer For TAD Applications
Zilog LPC Words Listing

Support Product Specifications

Z89C5900ZEM Emulation Module
Z89C6500ZDB Emulation Board
Z89C6501ZEM ICEBOX™ In-Circuit Emulator
Z89C6700ZDB Emulator Board
Z89C6700ZEM ICEBOX™ Emulator Board

Additional Information

Zilog's Superintegration™ Products Guide
Literature Ordering Guide
Zilog's Sales Offices, Representatives and Distributors

Z8® MICROCONTROLLERS - PERIPHERALS MULTIMEDIA FAMILY OF PRODUCTS

Databooks By Market Niche	Part No	Unit Cost
Digital Signal Processors Databook	DB95DSP0105	$ 5.00

Product Specification

- Z89321/371/391 16-Bit Digital Signal Processor

Application Notes

- Using the Z89321/371/391 CODEC Interface
- Z89321/371/391 Interprocessor Communication

Support Product Specification

- Z8937100ZEM ICEBOX™ In-Circuit Emulator -371

Additional Information

- General Terms and Conditions of Sale
- Zilog's Sales Offices, Representatives and Distributors
- Literature Guide and Ordering Information

Databooks By Market Niche	Part No	Unit Cost
Keyboard/Mouse/Pointing Devices Databook	DC-8304-01	$ 5.00

Product Specifications

- Z8602/14 NMOS Z8® 8-Bit Keyboard Controller
- Z8615 NMOS Z8® 8-Bit Keyboard Controller
- Z86C15 CMOS Z8® 8-Bit MCU Keyboard Controller
- Z86E23 Z8® 8-Bit Keyboard Controller with 8K OTP
- Z86C04/C08 CMOS Z8® 8-Bit Microcontroller
- Z86E08 CMOS Z8® 8-Bit Microcontroller
- Z88C17 CMOS Z8® 8-Bit Microcontroller
- Z86C117/717 Z8® 8-Bit Microcontroller
- Z86217 Z8® 8-Bit Microcontroller

Application Notes

- Z8602 Keyboard
- Z86C17 In-Mouse Applications

Support Product Specifications and Third Party Support

- Z0860200ZCO Evaluation Board
- Z0860200ZDP Adaptor Kit
- Z86C0800ZCO Evaluation Board
- Z86C0800ZDP Adaptor Kit
- Z86C1200ZEM Emulator
- Z86E2300ZDP Adaptor Kit
- Z86E2301ZDP Adaptor Kit
- Z86E2300ZDV Adaptor Kit
- Z86E2301ZDV Adaptor Kit

Additional Information

- Zilog's Superintegration™ Products Guide
- Literature Guide and Ordering Information
- Zilog's Sales Offices, Representatives and Distributors

Z8® MICROCONTROLLERS - PERIPHERALS MEMORY FAMILY OF PRODUCTS

Databooks By Market Niche	Part No	Unit Cost
Mass Storage Solutions	**DC-8303-01**	**$ 5.00**

Product Specifications

Z86C21 8K ROM Z8 CMOS Microcontroller
Z86E21 CMOS Z8 8K OTP Microcontroller
Z86C91 CMOS Z8 ROMless Microcontroller
Z86C93 CMOS Z8 Multiply/Divide Microcontroller
Z86C95 Z8 Digital Signal Processor
Z86018 Data Path Controller
Z89C00 16-Bit Digital Signal Processor

Application Note

Understanding Q15 Two's Complement Fractional Multiplication (Z89C00 DSP)

Support Product Specifications

Z8060000ZCO Development Kit
Z86C1200ZEM In-Circuit Emulator
Z86E2100ZDF Adaptor Kit
Z86E2100ZDP Adaptor Kit
Z86E2100ZDV Adaptor Kit
Z86E2101ZDF Conversion Kit
Z86E2101ZDV Conversion Kit
Z86C9300ZEM ICEBOX™ Emulator
Z86C9500ZCO Evaluation Board
Z8® S Series Emulators, Base Units and Pods
Z89C0000ZAS Z89C00 Assembler, Linker and Librarian
Z89C0000ZCC Z89C00 C Cross Compiler
Z89C0000ZEM In-Circuit Emulator -C00
Z89C0000ZSD Z89C00 Simulator/Debugger
ZPCMCIA0ZDP PCMCIA Extender Card

Additional Information

Zilog's Superintegration™ Products Guide
Zilog's Literature Guide
Zilog's Sales Offices, Representatives and Distributors

Z8 Technical Manuals and Users Guides	Part No.	Unit Cost
Z8® Microcontrollers User's Manual	UM95Z800103	5.00
Z86018 Preliminary User's Manual	DC-8296-00	N/C
Digital TV Controller User's Manual	DC-8284-01	5.00
Z89C00 16-Bit Digital Signal Processor User's Manual/DSP Software Manual	DC-8294-02	5.00
Z86C95 16-Bit Digital Signal Processor User Manual	DC-8595-02	5.00
Z86017 PCMCIA Adaptor Chip User's Manual and Databook	DC-8298-03	5.00
PLC Z89C00 Cross Development Tools Brochure	DC-5538-01	N/C

Z8® Application Notes	Part No	Unit Cost
The Z8 MCU Dual Analog Comparator	DC-2516-01	N/C
Z8 Applications for I/O Port Expansions	DC-2539-01	N/C
Z86E21 Z8 Low Cost Thermal Printer	DC-2541-01	N/C
Zilog Family On-Chip Oscillator Design	DC-2496-01	N/C
Using the Zilog Z86C06 SPI Bus	DC-2584-01	N/C
Interfacing LCDs to the Z8	DC-2592-01	N/C
X-10 Compatible Infrared (IR) Remote Control	DC-2591-01	N/C
Z86C17 In-Mouse Applications	DC-3001-01	N/C
Z86C40/E40 MCU Applications Evaluation Board	DC-2604-01	N/C
Z86C08/C17 Controls A Scrolling LED Message Display	DC-2605-01	N/C
Z86C95 Hard Disk Controller Flash EPROM Interface	DC-2639-01	N/C
Three Z8® Applications Notes: Timekeeping with Z8; DTMF Tone Generation; Serial Communication Using the CCP Software UART	DC-2645-01	N/C

Z80®/Z8000® DATACOMMUNICATIONS FAMILY OF PRODUCTS

Databooks By Market Niche	Part No	Unit Cost
High-Speed Serial Communication Controllers	DC-8314-01	5.00
Product Specifications		
Z16C30 CMOS Universal Serial Controller (USC™) (Preliminary)		
Z16C32 Integrated Universal Serial Controller (IUSC™) (Preliminary)		
Application Notes		
Using the Z16C30 Universal Serial Controller with MIL-STD-1553B		
Design a Serial Board to Handle Multiple Protocols		
Datacommunications IUSC™/MUSC™ Time Slot Assigner		
Support Products and Third Party Vendor Support		
Z16C3001ZCO Evaluation Board Product Specification		
Z16C3200ZCO Evaluation Board Product Specification		
Z8018600ZCO Evaluation Board Product Specification		
ZEPMDC00001 EPM™ Electronic Programmer's Manual Product Specification		
Third Party Vendors		
Additional Information		
Zilog's Superintegration™ Products Guide		
General Terms and Conditions of Sale		
Sales Offices, Representatives and Distributors		
Literature Guide		
Serial Communication Controllers	DC-8316-01	5.00
Product Specifications		
Z8030/Z8530 Z-Bus® SCC Serial Communication Controller		
Z80C30/Z85C30 CMOS Z-Bus® SCC Serial Communication Controller		
Z80230 Z-Bus® ESCC™ Enhanced Serial Communication Controller (Preliminary)		
Z85230 ESCC™ Enhanced Serial Communication Controller		
Z85233 EMSCC™ Enhanced Mono Serial Communication Controller		
Z85C80 SCSCI™ Serial Communications and Small Computer Interface		
Z16C35/Z85C35 CMOS ISCC™ Integrated Serial Communications Controller		
Application Notes		
Interfacing Z8500 Peripherals to the 68000		
SCC in Binary Synchronous Communications		
Zilog SCC Z8030/Z8530 Questions and Answers		
Integrating Serial Data and SCSI Peripheral Control on One Chip		
Zilog ISCC™ Controller Questions and Answers		
Boost Your System Performance Using the Zilog ESCC™		
Zilog ESCC™ Controller Questions and Answers		
The Zilog Datacom Family with the 80186 CPU		
On-Chip Oscillator Design		
Support Products		
Z8S18000ZCO Evaluation Board Product Specification		
Z8523000ZCO Evaluation Board Product Specification		
Z8018600ZCO Evaluation Board Product Specification		
ZEPMDC00002 Electronic Programmer's Manual Software		
Additional Information		
Zilog's Superintegration™ Products Guide		
Sales Offices, Representatives and Distributors		
Literature Guide		

Z80®/Z8000® DATACOMMUNICATIONS FAMILY OF PRODUCTS

Databooks	Part No	Unit Cost
Z80 Family Databook	**DC-8321-00**	**5.00**

Discrete Z80® Family

Z8400/C00 NMOS/CMOS Z80® CPU Product Specification
Z8410/C10 NMOS/CMOS Z80 DMA Product Specification
Z8420/C20 NMOS/CMOS Z80 PIO Product Specification
Z8430/C30 NMOS/CMOS Z80 CTC Product Specification
Z8440/C40 NMOS/CMOS Z80 SIO Product Specification

Embedded Controllers

Z84C01 Z80 CPU with CGC Product Specification
Z8470 Z80 DART Product Specification
Z84C90 CMOS Z80 KIO™ Product Specification
Z84013/015 Z84C13/C15 IPC/EIPC Product Specification

Application Notes and Technical Articles

Z80® Family Interrupt Structure
Using the Z80® SIO with SDLC
Using the Z80® SIO in Asynchronous Communications
Binary Synchronous Communication Using the Z80® SIO
Serial Communication with the Z80A DART
Interfacing Z80® CPUs to the Z8500 Peripheral Family
Timing in an Interrupt-Based System with the Z80® CTC
A Z80-Based System Using the DMA with the SIO
Using the Z84C11/C13/C15 in Place of the Z84011/013/015
On-Chip Oscillator Design
A Fast Z80® Embedded Controller
Z80® Questions and Answers

Additional Information

Zilog's Superintegration™ Products Guide
Literature Guide
Third Party Support Vendors
Zilog's Sales Offices, Representatives and Distributors

III

Z80®/Z8000® DATACOMMUNICATIONS FAMILY OF PRODUCTS

Databooks	Part No	Unit Cost
Z180™ Microprocessors and Peripherals Databook	**DC-8322-01**	**5.00**
Product Specifications		
Z80180/Z8S180/Z8L180 Z180™ Microprocessor		
Z80181 Z181™ Smart Access Controller (SAC™)		
Z80182/Z8L182 Zilog Intelligent Peripheral Controller (ZIP™)		
Application Notes and Technical Articles		
Z180™ Questions and Answers		
Z180™/SCC Serial Communication Controller Interface at 10 MHz		
Interfacing Memory and I/O to the 20 MHz Z8S180 System		
Break Detection on the Z80180 and Z181™		
Local Talk Link Access Protocol Using the Z80181		
Z182 Programming the MIMIC Autoecho ECHOZ182 Sample Code		
High Performance PC Communication Port Using the Z182		
Improving Memory Access Timing in Z182 Applications		
Support Products		
Z8S18000ZCO Evaluation Board		
Z8018100ZCO Evaluation Board		
Z8018101ZCO Evaluation Board		
Z8018101ZA6 Driver Software		
Z8018100ZDP Adaptor Kit		
Z8018200ZCO Evaluation Board		
ZEPMIP00001 EPM™ Electronic Programmer's Manual		
ZEPMIP00002 EPM Electronic Programmer's Manual		
Z80® and Z80180 Hardware and Software Support		
Additional Information		
Zilog's Superintegration™ Products Guide		
Literature Guide		
Zilog's Sales Offices, Representatives and Distributors		

Z80®/Z8000® DATACOMMUNICATIONS FAMILY OF PRODUCTS

Databooks and User's Manuals	Part No	Unit Cost
Z8000 Family of Products	**DC-8319-00**	**5.00**
Z8000 Family Databook		
Zilog's Z8000 Family Architecture		
Z8001/Z8002 Z8000 CPU Product Specification		
Z8016 Z8000 Z-DTC Product Specification		
Z8036 Z8000 Z-CIO Product Specification		
Z8536 CIO Counter/Timer and Parallel I/O Unit Product Specification		
Z8038/Z8538 FIO FIFO Input/Output Interface Unit Product Specification		
Z8060/Z8560 FIFO Buffer Unit		
Z8581 Clock Generator and Controller Product Specification		
User's Manuals		
Z8000 CPU Central Processing Unit User's Manual		
Z8010 Memory Management Unit (MMU) User's Manual		
Z8036 Z-CIO/Z8536 CIO Counter/Timer and Parallel Input/Output User's Manual		
Z8038 Z8000 Z-FIO FIFO Input/Output Interface User's Manual		
Z8000 Application Notes and Military Products		
Application Notes		
Using SCC with Z8000 in SDLC Protocol		
SCC in Binary Synchronous Communication		
Zilog's Military Products Overview		
Additional Information		
Zilog's Superintegration™ Products Guide		
Literature Guide		
Zilog's Sales Offices, Representatives and Distributors		
Z80 Family Microprocessor Family User's Manual	**DC-8309-01**	**5.00**
User's Manuals		
Z80 Central Processing Unit (CPU)		
Z80 Counter Timer Channels (CTC)		
Z80 Direct Memory Access (DMA)		
Z80 Parallel Input/Output (PIO)		
Z80 Serial Input/Output (SIO)		
Additional Information		
Zilog's Superintegration™ Products Guide		
Zilog's Sales Offices, Representatives and Distributors		
Literature Guide		

Databooks and User's Manuals	Part No	Unit Cost
Z80180 Z180 MPU Microprocessor Unit Technical Manual	DC-8276-04	5.00
Z280 MPU Microprocessor Unit Technical Manual	DC-8224-03	5.00
Z380™ Product Specification	DC-6003-03	N/C
Z380™ User's Manual	PS953800104	5.00
Z2000 Spread-Spectrum Transceiver Advance Information Product Specification	DC-6021-00	N/C
ZNW2000 PC WAN Adapter Board Development Kit User's Manual	UM95Z800101	N/C
SCC Serial Communication Controller User's Manual	DC-8293-02	5.00
High-Speed SCC, Z16C30/Z16C32 User's Manual	DC-8350-00	5.00

MILITARY COMPONENTS FAMILY

Military Product Specifications	Part No	Unit Cost
Z8681 ROMless Microcomputer	DC-2392-02	N/C
Z8001/8002 Military Z8000 CPU Central Processing Unit	DC-2342-03	N/C
Z8581 Military CGC Clock Generator and Controller	DC-2346-01	N/C
Z8030 Military Z8000 Z-SCC Serial Communications Controller	DC-2388-02	N/C
Z8530 Military SCC Serial Communications Controller	DC-2397-02	N/C
Z8036 Military Z8000 Z-CIO Counter/Timer Controller and Parallel I/O	DC-2389-01	N/C
Z8038/8538 Military FIO FIFO Input/Output Interface Unit	DC-2463-02	N/C
Z8536 Military CIO Counter/Timer Controller and Parallel I/O	DC-2396-01	N/C
Z8400 Military Z80 CPU Central Processing Unit	DC-2351-02	N/C
Z8420 Military PIO Parallel Input/Output Controller	DC-2384-02	N/C
Z8430 Military CTC Counter/Timer Circuit	DC-2385-01	N/C
Z8440/1/2/4 Z80 SIO Serial Input/Output Controller	DC-2386-02	N/C
Z80C30/85C30 Military CMOS SCC Serial Communications Controller	DC-2478-02	N/C
Z84C00 CMOS Z80 CPU Central Processing Unit	DC-2441-02	N/C
Z84C20 CMOS Z80 PIO Parallel Input/Output	DC-2384-02	N/C
Z84C30 CMOS Z80 CTC Counter/Timer Circuit	DC-2481-01	N/C
Z84C40/1/2/4 CMOS Z80 SIO Serial Input/Output	DC-2482-01	N/C
Z16C30 CMOS USC Universal Serial Controller (Preliminary)	DC-2531-01	N/C
Z80180 Z180 MPU Microprocessor Unit	DC-2538-01	N/C
Z84C90 CMOS KIO Serial/Parallel/Counter Timer (Preliminary)	DC-2502-00	N/C
Z85230 ESCC Enhanced Serial Communication Controller	DC-2595-00	N/C

GENERAL LITERATURE

Catalogs, Handbooks, Product Flyers and Users Guides	Part No	Unit Cost
Superintegration Master Selection Guide 1994-1995	DC-5634-01	N/C
Superintegration Products Guide	DC-5676-00	N/C
Quality and Reliability Report	DC-8329-01	N/C
ZIA™ 3.3-5.5V Matched Chip Set for AT Hard Disk Drives Datasheet	DC-5556-01	N/C
ZIA ZIA00ZCO Disk Drive Development Kit Datasheet	DC-5593-01	N/C
Zilog Hard Disk Controllers - Z86C93/C95 Datasheet	DC-5560-01	N/C
Zilog Infrared (IR) Controllers - ZIRC™ Datasheet	DC-5558-01	N/C
Zilog V. Fast Modem Controller Solutions	DC-5525-02	N/C
Zilog Digital Signal Processing - Z89320 Datasheet	DC-5547-01	N/C
Zilog Keyboard Controllers Datasheet	DC-5600-01	N/C
Z380™ - Next Generation Z80®/Z180™ Datasheet	DC-5580-02	N/C
Fault Tolerant Z8® Microcontroller Datasheet	DC-5603-01	N/C
32K ROM Z8® Microcontrollers Datasheet	DC-5601-01	N/C
Zilog Datacommunications Brochure	DC-5519-00	N/C
Z89300 DTC Controller Family Brochure	DC-5608-01	N/C
Zilog Digital Signal Processing Brochure	DC-5536-02	N/C
Zilog ASSPs - Partnering With You Product Brochure	DC-5553-01	N/C
Zilog Wireless Products Datasheet	DC-5630-00	N/C
Zilog Z8604 Cost Efficient Datasheet	DC-5662-00	N/C
Zilog Chip Carrier Device Packaging Datasheet	DC-5672-00	N/C
Zilog Database of IR Codes Datasheet	DC-5631-00	N/C
Zilog PCMCIA Adapter Chip Z86017 Datasheet	DC-5585-01	N/C
Zilog Television/Video Controllers Datasheet	DC-5567-01	N/C
Zilog TAD Controllers - Z89C65/C67/C69 Datasheet	DC-5561-02	N/C
Zilog Z87000 Z-Phone Datasheet	DC-5632-00	D/C
Zilog 1993 Annual Report	DC-1993-AR	N/C
Zilog 1994 Annual Report	DC-1994-AR	N/C