Chapter 8
Software Considerations and Program Examples

SOFTWARE CONSIDERATIONS AND PROGRAM EXAMPLES

A device with the versatility of the Am9513 is necessarily complex. This software application section simplifies the programmer's view of the Am9513 and facilitates quicker understanding and the implementation of the facilities provided. Hardware considerations are generally not discussed; indeed, with few exceptions they are deliberately excluded. For a detailed discussion of system implementation with the Am9513, the reader is referred to earlier chapters of this manual.

Example Languages

Software application notes generally provide specific examples of code written in one or more languages, structured as much to illustrate the purpose of the code clearly as well as to provide compact, working programs. Most languages have drawbacks, a fact that is of little importance in an application note, although it is desirable that examples should be presented in both high-level and assembly-level languages.

This software application note presents high-level constructs and examples in the C language, which now is seeing a fairly wide acceptance and is currently supported on systems supplied by Advanced Micro Devices. C is a block-structured language that, although lacking in features such as strong type checking, provides excellent bit-manipulation facilities in record fields, allowing such statements as:

```
#define OFF 1
Master.FOUT_gate = OFF;
```

This statement is part of a record initialization that flags the FOUT gate as being disabled. The "#define" statement assigns the value "1" to the constant identifier "OFF". The bit field ".FOUT_gate" of the record structure "Master" is then assigned the value of the identifier "OFF", i.e., "1". A similar Z8000™ assembler listing might take the form:

```
CONST   OFF = 0X1000  ; % AMD Mnemonics
LD   R4, OFF          ;
OR   R4, MASTER       ;
LD MASTER,R4          ;
```

Throughout this note several C terms have been redefined. Many programmers are familiar with languages other than C, possibly of the Pascal or PL/X variety, so the following definitions have been used in an attempt to achieve some common ground:

```
#define BEGIN
#define END
#define RECORD   struct
#define THEN
```

These definitions allow the use of the type identifier "RECORD" for structure definitions rather than the normal C type identifier "struct".

Assembly language examples are presented using Am8080/Am8085, Z80 and Z8000 source mnemonics having formats compatible with the AMD or Zilog assemblers. Macros are used wherever it is felt that they aid the programmer, mostly within the Am9513 command structure. The macros are generally optimized, such that better code can not be generated by hand, and mostly employ strict parameter checking (eliminating excuses for not using them). For example, to save counters 3, 4 and 5 would be accomplished by the statement:

```
SAVE  3,4,5
```

Code generated by this statement would be (Am8080/Am8085):

```
MVI     A,0BCH
OUT     CONTROL
```

Macros are presented for setting the Counter registers and the Master register and, although the experienced Am9513 user may find them of dubious use, the novice will appreciate the code generated. For example, to set the Master register, the following command may be used:

```
MASTER  TOD_50HZ,DISABLE,DISABLE,GATE_1,
        11,ON,BUS_8,ON,BCD
```

The following code sequence will be generated (Am8080/Am8085):

```
MVI     A,17H
OUT     CONTROL         ;  Point to Master reg
MVI     A,61H
OUT     DATA            ;  Send Low command byte
MVI     A,8BH
OUT     DATA            ;  High command byte
```

By examining the code generated (as above) by the macro assembler, the user may gain quick understanding of the usage of the Am9513. (Notice that the above assembly code does not use intermediate storage records for the data. Use of assembly language often implies high efficiency or compact code is required.) C examples have been compiled to Am8080/Am8085 code using the Whitesmith's C compiler and Z8000 code using the AMD C cross compiler, both running on the AMD System

8/8. All executable Am8080/Am8085 and Z80 target code has been tested using an AMD System 8/8 with an Am95/5032 ROM/EPROM board installed, supporting an Am9513 with control and data ports decoded at I/O addresses DAH and D8H, respectively. All executable Z8000 code has been tested using the above system with an Am96/4116 Z8000 bus master card installed.

FUNCTIONAL DESCRIPTION

(See Chapter 1 for detailed description.)
The Am9513 includes five general purpose 16-bit counters. A variety of internal frequency sources and external pins may be selected as inputs for individual counters with software selectable active-high or active-low input polarity. Hardware gating of each counter is available. Each counter provides either pulsed or level as well as tri-state and fixed low outputs. The counters can be individually programmed to count up or down in BCD or binary modes. The accumulated count may be read without disturbing the counting process. Any of the counters may be internally concatenated to form an effective counter length of up to 80 bits.

Associated with each counter are a Load register and a Hold register. The Load register automatically reloads the counter to a predefined 16-bit value, thus controlling the effective count period. The Hold register acts either as a second 16-bit Load register for complex waveform generation or as a 16-bit storage register to save count values without disturbing the counting process, thereby permitting the host processor to read intermediate count values.

Two counters have additional Alarm registers and comparators with associated logic to allow operation in a 24-hour time-of-day mode with alarm facility. Clocking may be either in real time or programmed over the full dynamic clocking range of the Am9513.

Each of the five counters has a dedicated output pin that may be programmed to provide a variety of outputs. General-purpose counter inputs are available for configuration under software control, allowing dynamic reassignment of inputs with the facility, for example, to use a single gate pin simultaneously as a clock input to one counter and as a gate input to another.

HARDWARE CONSIDERATIONS

Prefetch

In order to minimize the read access time to internal Am9513 registers, a prefetch circuit is used for all read operations through the Data port. Following each read or write operation through the Data port, the Data Pointer register is updated to point to the next register to be accessed. Immediately following this update, the new register data is transferred to a special prefetch latch at the interface pad logic. When the user performs a subsequent read of the Data port, the data bus drivers are enabled, outputting the prefetched data on the bus. Since the internal data register is accessed prior to the start of the read operation, its access time is transparent to the user. In order to keep the prefetched data consistent with the Data Pointer, prefetches are also performed after each write to the Data port and after execution of the "Load Data Pointer" command. The following rules for Data port Transfers should be heeded:

1. The Data Pointer register should always be reloaded before reading from the Data port if a command other than "Load Data Pointer" (point_to or POINT) was issued to the Am9513 following the last Data port read or write. The Data Pointer does not have to be loaded again if the first Data port transaction after a command entry is a write, since the Data port write will automatically cause a new prefetch to occur.
2. Operating modes N, O, Q, R and X allow the user to save the counter contents in the Hold register by applying an active-going gate edge. If the Data Pointer register had been pointing to the Hold register in question, the prefetched value will not correspond to the new value saved in the Hold register. To avoid reading an incorrect value, a new "Load Data Pointer" command should be issued before attempting to read the saved data. A Data port write (to another register) will also initiate a prefetch; subsequent reads will access the recently saved Hold register data. Many systems use the "saving" gate edge to interrupt the host CPU. In such systems, the interrupt service routine should issue a "Load Data Pointer" command prior to reading the saved data.

Memory Mapping

Associated with the Am9513 are four parameters known collectively as read/write recovery times. Certain coding sequences can violate these parameters in sometimes non-obvious ways. Consider a Motorola 6800-based system, employing memory-mapped input/output. Reading the hold register on counter 4 may be accomplished by the following code sequence:

```
LDAA    # 14       Point to counter 4 hold
STAA    CONTROL
LDX     DATA       Read the register
STX     RESULT     Store the data
```

Notice that the operation "LDX" performs two 8-bit reads from location DATA and DATA+1 using two consecutive clock cycles. For a 1250ns read recovery time the maximum system clock frequency allowable without violating the read recovery time is 680kHz. This is lower than many system clocks in use, so the following code sequence may be used to avoid this limitation:

```
LDAA    # 14       Point to counter 4 hold
STAA    CONTROL
LDAA    DATA       read low byte
STAA    RESULT     reversed order for
                   compatability with above
LDAA    DATA+1     read high byte
STAA    RESULT
```

The equivalent multiple write operation is even more restrictive: an 1800ns write recovery time limiting the system clock to 500kHz unless the separate data byte write technique is adopted.

Similar considerations apply for memory-mapped Am8080/Am8085 systems. Notice that in the case of normal I/O mapped Z80 and Z8000 systems, the block input/output instructions may

also violate the read/write recovery time parameters at very high system clock rates. Using the normal input/output instructions avoids violations without recourse to the use of wait states.

DATA MODEL

The first task of the programmer is to construct a data model of the Am9513 around which the applications software may be draped. A topdown, structured approach is adopted and although for purposes here the data model need not be optimized either for space utilization or for access time, both aspects are in fact efficiently implemented.

Figure 8-1 shows the data model to be adopted, which should be contrasted with Figures 8-2, 8-3 and 8-4 depicting the hardware structure being modeled. Notice that the Command register and Data Pointer register do not appear in the data model.

The simplicity of Figure 8-1 illustrates that the Am9513 is singularly well suited to this approach; removing extraneous internal hardware aspects focuses the attention of the programmer on the salient details.

C provides a facility called "typedef" for creating new data type names, similar to the "TYPE" facility in other languages. For example, the declaration

```
typedef int LENGTH;
```

makes the name 'LENGTH' a synonym for 'int'. The type 'LENGTH' can be used wherever the type 'int' can be used. Similarly, the two record definitions of Figure 8-1 can be replaced by their respective type names; indeed, the definition of

```
                                    /* Single counter set          */
typedef RECORD      (count_type     mode ;
                     unsigned int   load  ;
                     unsigned int   hold  ;
                    ) channel_type ;

                                    /* Am9513 chip set             */
typedef RECORD      (master_type    master ;
                     channel_type   counter [5] ;
                     status_type    status ;
                     unsigned int   alarm_1 ;
                     unsigned int   alarm_2 ;
                    ) Am9513_type ;
```

Figure 8-1. Software Structure of the Am9513

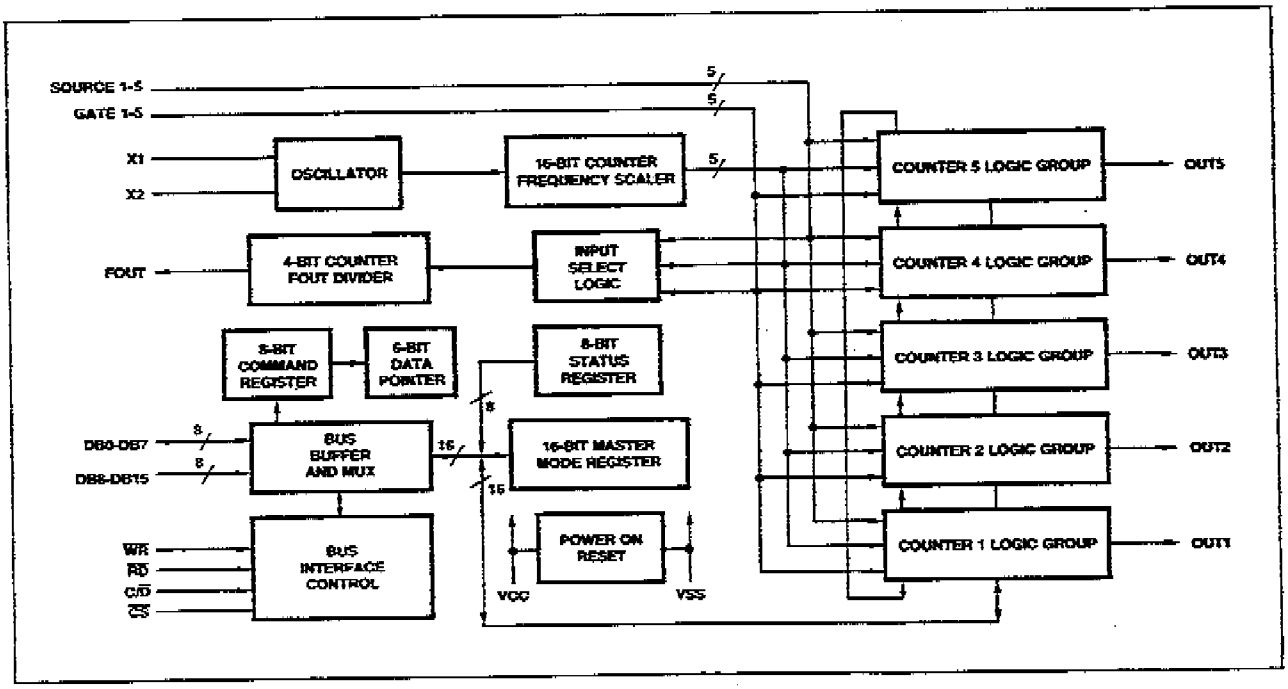

Figure 8-2. General Block Diagram

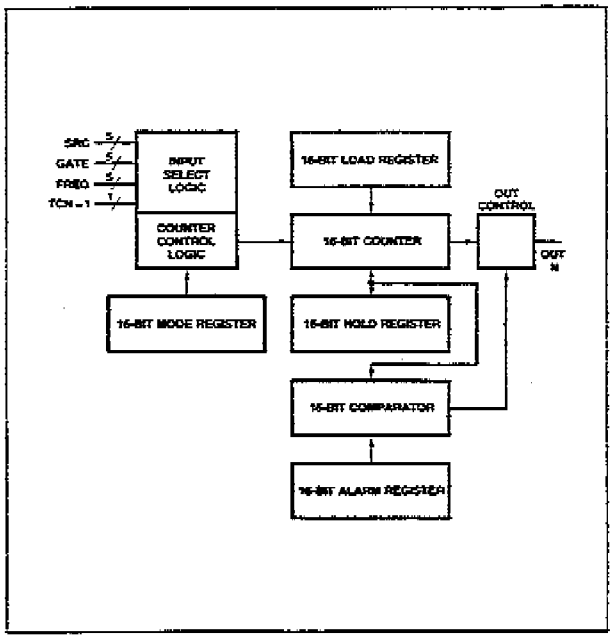

Figure 8-3. Counter Logic Groups 1 and 2

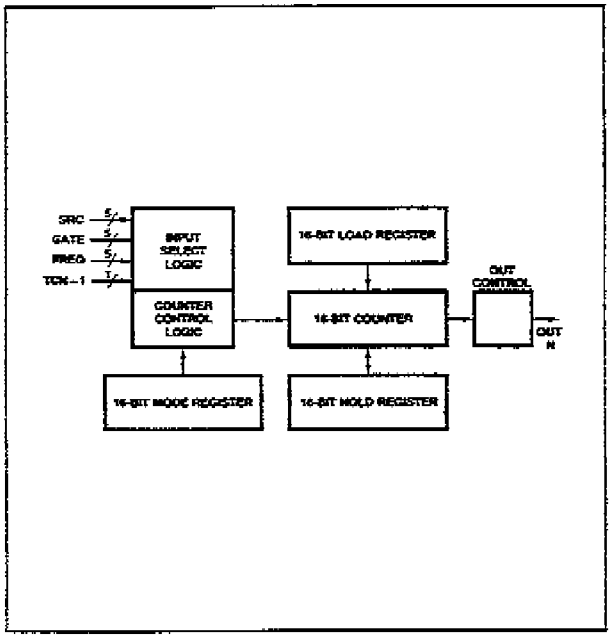

Figure 8-4. Counter Logic Groups 3, 4 and 5

the 'Am9513_type' uses the type 'channel_type' as a field to identify the array of five counters on the Am9513. Notice from Figure 8-1 that each counter (or channel_type) consists of three registers known as the Mode, Load and Hold registers. Additionally the Mode register itself is further defined via a typedef in Figure 8-6 to consist of a series of bit fields. The sum of the field widths of these bit fields is 16, that is, the Mode register is 16 bits wide.

It must be emphasized that a C typedef declaration does not create a new type in any sense (unlike the TYPE declaration in other languages); it merely adds a new name for some existing type. The use of typedef's makes C programs more portable and significantly improves readability.

An application using an array of 20 Am9513 devices with an access pointer may be declared as shown in Figure 8-5a.

```
Am9513_type
                    /* recall we have a new "type"*/
Am9513[20],         /* data structure for 20 chips*/
*Am9513_ptr         ;  /* access pointer */
```

Figure 8-5a.

Notice that C uses array subscripts commencing with zero; the third Am9513 device would be referred to by Am9513[2]. Should the reader prefer, all arrays can be declared with a dummy entry, such that all zero indices may be ignored as shown in Figure 8-5b.

```
Am9513 type
Am9513[21],  /* 21 devices, ignore device #0 */
```

Figure 8-5b.

The third Am9513 device may now be referenced by the form Am9513[3]. Throughout this software application note the first form (zero subscript allowed) will be employed except where clearly stated otherwise.

The following discussions of the data model fields (the various hardware registers on the Am9513) are accompanied by examples of usage. The examples refer to the model, which is transferred to the hardware device via some input/output operations. Such operations are discussed with the data pointer sequencing facility description, and are omitted from most other C examples for clarity.

Users of other languages should find the data model and C examples useful as an aid to understanding the Am9513. Assembly language examples are included for additional clarity.

COMMAND REGISTER

The Command register provides direct control over each of the five general counters (via the Control port) and controls access to the counter registers (via the Data port) by updating the Data Pointer register. Commands are instruction codes to the Am9513 and as such are not part of the data model. They are generally used in the form:

```
eg ( )
BEGIN
        int reset = 0XFF,          /* C code */
            load_all_counters = 0X5F;
    output (CONTROL, & reset);
    output (CONTROL, & load_all_counters);
END
```

Notice that the variables "reset" and "load_all_counters" are declared AUTO (LOCAL in some languages) such that they reside on the stack while the procedure "e.g.()" is current. The operation of the procedure "output (X,Y)" will be described later in this section; suffice to say that the address of the variable "Y" to be output (sent to the Am9513) is passed as one parameter and the destination port "X" (Control or Data) of the Am9513 as the other. Thus the variable "Y" is called by reference (e.g., & reset) and the destination port "X" is called by value (e.g., Control).

```
                        ; Z80 Macro Assembler code
ENTRY:
        RESET           ; Macro simplicity
        RET
```

The "Command Description" section later in this chapter explains the detailed operation of commands available with example usage.

DATA POINTER REGISTER

The Data Pointer register is a write-only register controlled solely by a command with the structure illustrated in Figure 8-6. The detailed hardware format of the register is irrelevant for purposes here since the data pointer command provides all information necessary. As a command, it does not appear in the data model.

```
/* Load data ptr register command */
typedef RECORD      (unsigned group          :3;
                     unsigned element        :2;
                     unsigned cmnd_code      :3;
                     )data_type ;
```

Figure 8-6. Load Data Pointer Command Structure

The data pointer command selects which internal register is to be accessible via the Data port and consists of a constant command code (000), a 2-bit element field and a 3-bit group field. (See Page 1-5 for hardware description.) Random access to any available internal register location can be accomplished by simply sending the appropriate data pointer command to the Control port and then performing a Data read or write. Sequential access to groups of internal registers may be performed by sending the appropriate enable and data pointer commands to the Control port and performing multiple Data reads and/or writes.

For example, random access to the Load register of counter 4 may be performed as shown in Figure 8-7. Sequential access to the hold registers of all five counters is performed as shown in Figure 8-8. Suitable C input/output routines for an 8-bit data bus may be defined as shown in Figure 8-9.

```
                                                    /* C example */
eg ( )
BEGIN
        point_to (4,LOAD) ;                         /* set up Data Pointer Register */
        Am9513.counter[3].load = input(DATA) ;      /* Read in the data to appropriate
                                                           field */
END
                                                    /* C utility */
point_to (channel, reg)                             /* set Data Pointer Register to
        unsigned int channel, reg ;                          channel,reg */

BEGIN
        data_type data_ptr ;                        /* local command structure */
        data_ptr.group = channel ;
        data_ptr.element = reg ;
        data_ptr.cmnd_code = 0 ;
        output (CONTROL,&data_ptr) ;                /* send Load Data Pointer Command */
END

                                    ; Z80 Macro example
STORE DEFS 2                        ; Destination of data
ENTRY:
        POINT   4,LOAD_             ; Point to counter 4 LOAD register
        LD      C,DATA              ; Set up port address
        LD      HL,STORE            ; Set up data destination
        INI                         ; Low byte of LOAD register
        INI                         ; High byte of LOAD register
        RET
```

Figure 8-7. Random Access to Registers

```
                                        /* C example */
eg ()
BEGIN
        int index = -1 ;                /* counter index, 0 to 4 */

        sequence (ENABLE) ;             /* Turn on data pointer seq */
        point_to (1,HOLD_CYCLE) ;       /* Set Data Pointer register */
        while ((index += 1) <5)         /* Counters 0 thru 4 */
                Am9513.counter[index].hold = input(DATA) ; /* 16 bit transfer */
END

sequence (request)                      /* C utility */
        int request ;
BEGIN
        int enable = OXE0 ,             /* commands for data pointer sequencing */
          disable = OXE8 ;

        if (request = = ENABLE)
                output (CONTROL, & enable) ;
        else
                output (CONTROL, & disable)
END
                                        ; Z80 Macro example
ENTRY:
        DPS ON                          ; Enable data pointer sequencing
        POINT 1,HOLD_CY                 ; Data pointer to hold cycle
        LD HL,HOLDS                     ; Data area
        LD B,5*2                        ; Byte count, 5 holds, 2 bytes each
        LD C,DATA                       ; Am9513 data port
        INIR                            ; Perform transfer
        RET
HOLDS:  DEFS 10                         ; Data area
                                        ; Z80 Macro example
ENTRY:
        DPS ON                          ; Enable data pointer sequencing
        POINT 1,HOLD_CY                 ; Data pointer to hold cycle
        LD HL,HOLDS                     ; Data area
        LD B,5*2                        ; Byte count, 5 holds, 2 bytes each
        LD C,DATA                       ; Am9513 data port
        INIR                            ; Perform transfer
        RET
HOLDS:  DEFS 10                         ; Data area
                                        ; Am8080/Am8085 Macro example
        DPS     ON                      ; Enable data pointer sequencing
        POINT   1,MODE_                 ; Mode register 1
        LXI     H,ARRAY                 ; Storage area
        MVI     B,5*3*2                 ; 5 counters, 3 x 16 bit registers
LOOP:
        IN      DATA                    ; Get a byte
        MOV     M,A                     ; Save a byte
        INX     H                       ; Move the byte pointer
        DEC     B
        JNZ     LOOP
        RET
ARRAY:
MODE1 DS        2                       ; counter 1 data area
LOAD1 DS        2
HOLD1 DS        2
...
MODE5 DS        2                       ; counter 5 data area
LOAD5 DS        2
HOLD5 DS        2
```

Figure 8-8. Sequential Access to Registers

```
                                        /* C example */
unsigned int input (port)               /* Read 2 bytes from port */
        int port ;
BEGIN
        unsigned int temp;              /* local parameter to assemble word */
        temp = in (port) ;              /* Get low byte */
        temp += in (port) *256 ;        /* Put high byte into temp */
        return (temp) ;                 /* Return the 16 bit value read */
END
output (port, data)                     /* 2 bytes to DATA port or 1 byte */
      unsigned port, *data;             /* to CONTROL port (8 bit bus) */
BEGIN
        out (port, (*data %256));
        if (port == DATA)               /* 16 bit transfer to DATA port */
          out (port, (*data/256));
END
```

Figure 8-9. Input/Output Routines for 8-Bit Data Bus

```
typedef RECORD      (unsigned day_mode          :2;
                     unsigned compar_1          :1;
                     unsigned compar_2          :1;
                     unsigned FOUT_source       :4;
                     unsigned FOUT_divisor      :4;
                     unsigned FOUT_gate         :1;
                     unsigned data_bus          :1;
                     unsigned data_ptr          :1;
                     unsigned scaler            :1;
                    )master_type;
```

Figure 8-10. Master Mode Register – Software Structure

MASTER MODE REGISTER

The 16-bit Master Mode Register controls those internal activities that are not controlled by the individual counter registers. Figure 8-10 shows the record fields of the 'Master-type' structure. Figure 8-11 illustrates allowable field values.

The individual field declarations show, for example, that the '.FOUT_source' field is four bits wide. Notice that the sum of all the field widths is 16; the Master Mode Register is 16 bits wide.

C compilers should be checked to ensure bit field declarations are implemented with first field at the least significant bit address. These definitions are correct for the AMD and Whitesmith's compilers; some other compilers may need the field order reversed.

After power-on reset or a Software Reset command the Master Mode Register is set to the following configuration (all field values zero):

```
.day-mode       = TOD_OFF
.compar_1       = DISABLE
.compar_2       = DISABLE
.FOUT_source    = F1
.FOUT_divisor   = 16
.FOUT_gate      = ON
.data-bus       = BUS_8
.data-ptr       = ON
.scaler         = BINARY
```

Notice that changing the FOUT status by altering the source divisor or gate fields may generate transients.

Time-of-Day

The .day_mode field can be turned off, allowing counters 1 and 2 to function the same way as counters 3, 4 and 5, or set to TOD_50Hz, TOD_60Hz or TOD_100Hz allowing a 24-hour clock function to be used. Refer to the software examples (in Chapter 4) for further information.

Comparators

Comparator registers exist for counters 1 and 2. If a comparator is enabled (e.g., .compar_1 = ENABLE), its output is substituted for the associated counter output. The output will remain active while the comparison is true. The two comparators can always be used individually in any operating mode. One special case occurs when the Time-of-Day option is invoked and both comparators are enabled. The operation of Comparator 2 will then be conditioned by Comparator 1 so that a full 32-bit compare must be true in order to generate a true signal on Output 2. Output 1 will continue, as usual, to reflect the state of the 16-bit comparison between Alarm 1 and Counter 1.

FOUT Source

The '.FOUT_source' field specifies the source input for the FOUT divider. Fifteen inputs are available for selection including the five source pins (SRC1-SRC5), the five Gate pins (GATE1-GATE5) and the five internal frequencies derived from the master oscillator (F1-F5).

```
e.g.,  Am9513.master.FOUT_source = GATE_4;
```

FOUT Divider

The '.FOUT_divisor' field specifies the dividing ratio for the FOUT divider. The .FOUT_source is divided by an integer value (1-16) and passed to the FOUT output buffer.

FOUT Gate

The '.FOUT_gate' provides a software gating facility for the FOUT output signal (ON or OFF). Notice that commands exist to directly gate the FOUT output on and off without using the Master register fields (e.g., FOUT ON).

Bus Width

The 'data_bus' field controls the width of the data bus interface by configuring the part for an 8-bit or 16-bit external data bus. Notice that the CP/M-compatible version of the Am9513 evaluation program and the Am8080/Am8085 and Z80 macros only

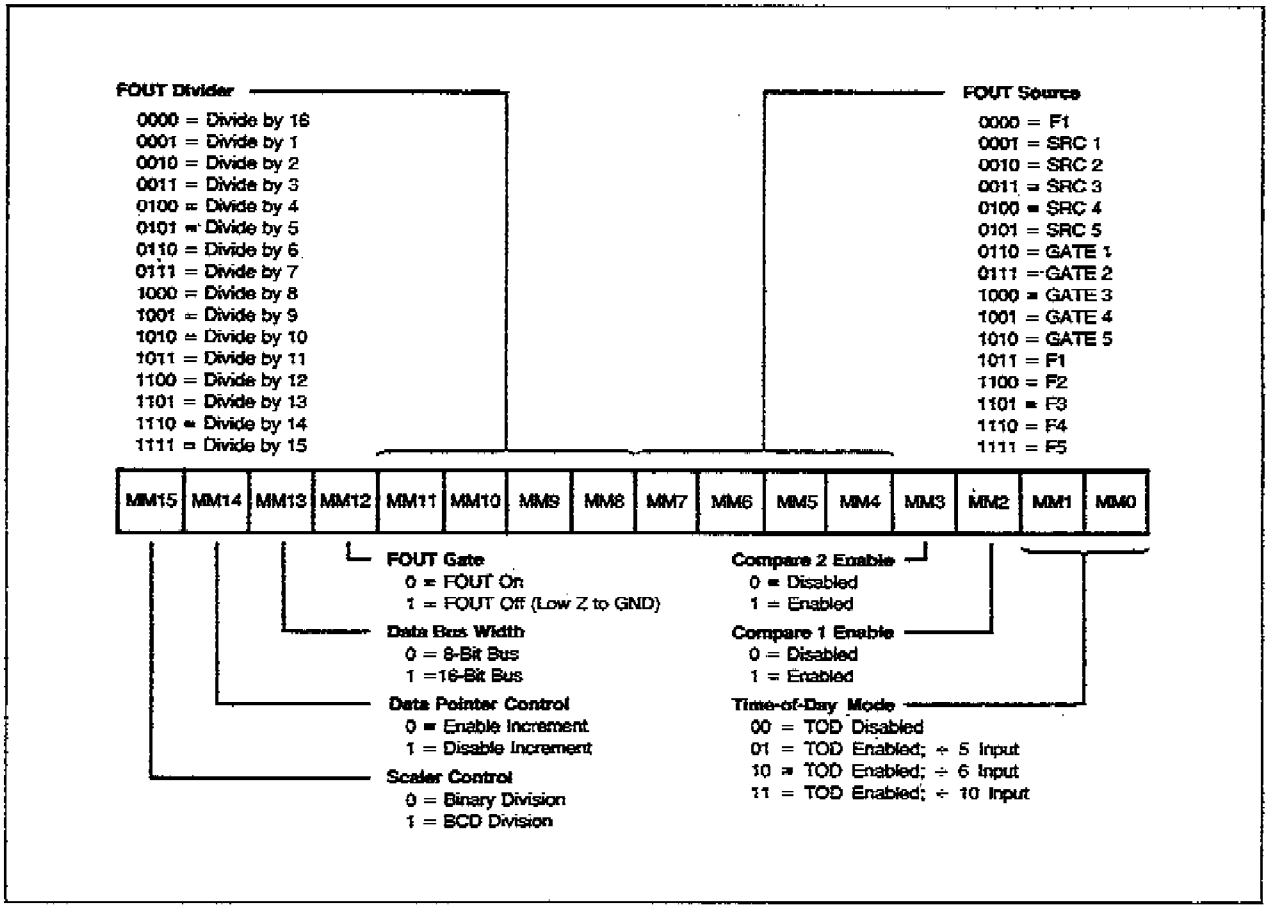

Figure 8-11. Master Mode Register – Hardware Structures

allow an 8-bit data bus configuration while the Z8000 macros only allow a 16-bit configuration. These constraints may be changed by the user. Notice that the macros require the bus width to be passed as a parameter, although it is ignored.

Data Pointer Sequencing

The '.data_ptr' field enables or disables the automatic sequencing functions, described under the Data Pointer Register section. Commands exist to directly enable or disable this function (e.g., DPS ON).

Scaler Ratio

The '.scaler' field controls the counting configuration of the frequency scaler counter. This configuration may be BCD or BINARY.

COUNTER REGISTERS

Counter Mode Register

The Counter Mode register configures the individual counters for various operating conditions. The Counter Mode register software structure is shown in Figure 8-12. (For hardware structure see Page 1-23.) On power-up or after Software Reset the Counter Mode registers are set to the following (equivalent) configurations:

```
                              /* C example */
.output      = OFF_LO_TC
.direction   = DOWN
.base        = BINARY
.control     = MODE_ABC
.source      = F1
.edge        = RISE
.gate        = NO_GATE
                              ; Macro example
MODE_REG 4,OFF_LO_TC,DOWN,BINARY,MODE_ABC,
                    F1,RISE,NO_GATE
```

Refer to Chapter 1, Figure 1-16 and 1-17 for detailed descriptions of the various modes available.

Load/Hold Registers

The counter Load register provides a base value for the counter. The counter Hold register provides either a second base register or a storage register for intermediate count values. The registers may be accessed as follows (Counter 4 illustrated, setting Load register to 4000 Hex):

```
                              /* C example */
Am9513.counter[3].load = 0X4000
                              ; Macro example
LOAD_REG   4,4000H
```

COMMAND DESCRIPTION

The Macro files and Macro command summary provide a detailed syntax and description of the action of the various command sequences and the commands available (Appendices D through G). For further details of the various modes of operation and command interaction refer to Page 1-25.

The following descriptions of the macro commands each provide an example usage. For multiple register commands, counters 2, 3 and 4 are used. For single register commands, counter 3 is used.

RESET		Issues a 'RESET' and 'LOAD' 1, 2, 3, 4, 5' command sequence, the latter command to ensure no counters are in the TC state. Refer to the register descriptions for the initial settings caused by a RESET. The Z8000 RESET Macro additionally sets the data bus width to 16 bits and issues a dummy load data pointer command (POINT 1, MODE_).
ARM	2, 3, 4	Issues an ARM command for up to 5 counters. Enables the listed counters to count source pulses. In modes G-L the next TC causes the counter to reload from the Hold register; in all other modes the next TC causes the counter to reload from the Load register.

```
                                                  /* Counter Mode Register */
typedef RECORD      (unsigned output              :3;
                     unsigned direction           :1;
                     unsigned base                :1;
                     unsigned control             :3;
                     unsigned source              :4;
                     unsigned edge                :1;
                     unsigned gate                :3;
                    )count_type;
```

Figure 8-12. Counter Mode Register -- Software Structure

OAD	2, 3, 4	Issues a LOAD command for up to 5 counters. Causes the listed counters to be loaded with the Load or Hold register depending on the Mode register and the state of the count cycle. If a listed counter is in the TC state, the counter counts once to leave TC, before loading, regardless of whether the counter is armed or the state of any gate input. If a listed counter is in the cycle preceding TC, the counter immediately goes to TC, regardless of whether the counter is armed or of the state of any gate input.
LD__ARM	2, 3, 4	Issues a LOAD AND ARM command for up to 5 counters. Operation is identical to issuing separate LOAD and ARM commands. If a listed counter is in the TC state the counter will count once to leave TC before loading and arming, regardless of whether the counter is armed or of the state of any gate input. If a listed counter is in the cycle immediately preceding TC the counter will immediately go to TC regardless of whether the counter is armed or of the state of any gate input. Avoid this command, by using separate LOAD and ARM commands where possible.
DISARM	2, 3, 4	Issues a DISARM command for up to 5 counters. If a listed counter is in the TC state, the counter counts one source edge, to leave TC, before disarming. Once disarmed all counting ceases.
SAVE	2, 3, 4	Issues a SAVE command for up to 5 counters. Transfers contents of listed counters into associated HOLD register independent of, and without affecting counter status.
DRMSAV	2, 3, 4	Issues a DISARM AND SAVE command for up to 5 counters. Identical to issuing a DISARM and a SAVE command.
SET__	3	Issues a SET command to a single counter. The output for the counter is driven high provided TC toggle mode is specified, otherwise nothing happens.
CLEAR	3	Issues a CLEAR command to a single counter. The output for the counter is driven low provided TC toggle mode is specified, otherwise nothing happens.
STEP	3	Issues a STEP command to a single counter. Increments or decrements the counter irrespective of armed status or gate conditions. The step direction depends on the Mode register. The results of stepping an armed counter while counting are undefined.
FOUT	ON	Issues a GATE ON FOUT or GATE OFF FOUT command. The FOUT output becomes active/inactive. Notice that a single transient pulse may be produced.
DPS	ON	Issues an ENABLE/DISABLE DATA POINTER SEQUENCE command. See the description of the Data Pointer register for further details.
POINT	3, MODE__	Issues a LOAD DATA POINTER REGISTER command with group, element information. See the description of the Data Pointer register for further details.
MASTER	TOD__OFF, DISABLE, DISABLE, F4, 15, ON, BUS__8, ON, BINARY	Issues a command sequence to set the Master register. The nine parameters correspond to the nine data model fields and as such are reversed in order, reading low order bits to high order bits from left to right. Notice that the nine parameters are NOT checked by the macros for space reasons.
MODE__REG	3, TC__TOGGLE, UP, BCD, MODE__ABC, F4, FALL, NO GATE	Issues a command sequence to set the Mode register for a counter. The first parameter identifies the counter and the next seven parameters correspond to the seven data model fields. The seven data model fields read low order bit to higher order bit from left to right. The seven parameters are NOT checked by the macros for space reasons.
LOAD__ REG	3, 4000H	Issues the command sequence to set the Load register for a single counter to a given value, in this case a constant, 4000H.
HOLD__ REG	3, 4000H, !	Issues the command sequence to set the Hold register of a single counter to a given value, in this case the contents of the address 4000H. The Z8000 macro does not require the indirection flag (!) since the AMD Assembler Macz is able to identify the indirection mode from context.

EXAMPLE: Am9513 EVALUATION PROGRAM

The Am9513 evaluation program is menu-driven allowing full functional testing and evaluation of an Am9513 located within a host system I/O address space. The evaluation program is written entirely in C and is provided under the CP/M (ver. 2.2) compatible AMDOS operating system to allow interactive usage without the necessity of a resident C compiler.

The Am9513 driver uses the data model and commands described earlier, thus providing a useful example of Am9513 C programming. Additionally, at certain points throughout the driver program, further explanation is provided on request, mostly in the area of the various operating modes available. All functions associated with the Am9513 may be exercised, including defining the port addresses associated with the device under test. However, since the target code is for the Am8080/Am8085, the data bus width is restricted to 8 bits.

Figure 8-13 provides an example of the coding used within the evaluation program and Figure 8-14 shows an actual user run-time test session (user input is highlighted).

```
Am9513_type
        Am9513,                         /* declare the storage */
        *Am9513_ptr ;

read_element (mode)

/*      This procedure will read the chip registers using either
SEQUENCE or random-access depending on the mode parameter request.
*/

        int mode ;
BEGIN
        int loop_count ;
        unsigned int A[3] ;                     /* temp array for register values */
        int group, element ;                    /* indices  */

        if ( mode = = SEQUENCE )
        THEN
        BEGIN
                sequence (ENABLE) ;
                point_to (1,MODE) ;
                Am9513_ptr = &Am9513 [0] ;
                loop_count = 0 ;
                while ( ( loop_count += 1 ) <6)
                BEGIN
                        Am9513_ptr->counter[group].mode = input (DATA) ;
                        Am9513_ptr->counter[group].load = input (DATA) ;
                        Am9513_ptr->counter[group].hold = input (DATA) ;
                        Am9513_ptr++ ;
                END
        END
        else
        BEGIN
                Am9513_ptr = &Am9513[0] ;
                group = 0 ;
                sequence (DISABLE) ;
                while ( ( group += 1 ) <6 )
                BEGIN
                        element = -1 ;
                        while ( ( element += 1 ) <4 )
                        BEGIN
                                point_to (group, element) ;
                                A [element] = input (DATA) ;
                        END
                        Am9513_ptr->counter[group].mode = A [0] ;
                        Am9513_ptr->counter[group].load = A [1] ;
                        Am9513_ptr->counter[group].hold = A [2] ;
                        Am9513_ptr++ ;
                END
        END
END
```

Figure 8-13. Am9513 Evaluation Program Segment

Main Menu

1. Set data bus width
2. Software reset
3. Set output channel
4. Set Master register
5. Set port addresses
6. Arm counters
7. Load
8. Load and arm
9. Disarm
A. Save
B. Disarm and save
C. Step
D. Read registers, etc
E. Set counter operating modes, etc
F. Clear output channel

<enter option>
D

0. All regs using data ptr sequencing
1. All regs using random access
2. More info
3. Any regs, random access
4. Any regs, random access, n times
5. Print entire 9513 reg. set
6. Return to main menu

<enter option>
0

	Last written					Just read				
#	mode			load	hold	mode		load	hold	TC
1	00001011	00000000	A	0H	0H	00000000	00000000	0H	0H	0
2	00001011	00000000	A	0H	0H	00000000	00000000	0H	0H	0
3	00011100	11101001	V	100H	200H	00000000	00000000	0H	0H	0
4	00001011	00000000	A	0H	0H	00000000	00000000	0H	0H	0
5	00001111	00100010	D	400H	0H	00000000	00000000	0H	0H	0

0. All regs using data ptr sequencing
1. All regs using random access
2. More info
3. Any regs, random access
4. Any regs, random access, n times
5. Print entire 9513 reg. set
6. Return to main menu

<enter option>
6

Main Menu

1. Set data bus width
2. Software reset
3. Set output channel
4. Set Master register
5. Set port addresses
6. Arm counters
7. Load
8. Load and arm
9. Disarm
A. Save
B. Disarm and save
C. Step
D. Read registers, etc
E. Set counter operating modes, etc
F. Clear output channel

<enter option>
6

Arm counters: enter 1-5 on one line
35
Code used 34H

Figure 8-14. Am9513 Evaluation Program Session