

Quasar Electronics

SX-II

8031 Single Board Computer

MVS-31 System Monitor

User's Manual

Written by

MicroValley Systems

Contents

Introduction	2
MVS-31 Monitor/Debugger Features	2
Getting Started.....	2
Monitor commands and conventions	2
Command Set Summary	3
Monitor Commands - Description	4
DA - DisAssemble program.....	4
DB - Display Breakpoints.....	4
DI - Display Internal data memory	4
DX - Display eXternal data memory	5
DR - Display Registers	5
DS - Display Special function registers	5
DT - Display Time and date	5
DW - Display Watchpoints.....	5
MI - Modify Internal data memory	6
MX - Modify eXternal data memory	6
ST - Set Time and date	6
<space> - Single step.....	6
N - Next in-line instruction single stepping	7
G - Go from breakpoint	7
J - Jump to user program	7
E - Enter breakpoint.....	7
C - Clear breakpoint	7
W - enter Watchpoint	8
K - Kill watchpoint	8
V - View watchpoint data	8
S0 - Load a Motorola S19 format file.....	8
S1 - Load a Motorola S19 format file.....	8
: - Load an Intel HEX format file.....	9
Register Modify commands.....	9
User Accessible Input/Output Functions	10
Input/Output Functions and Interrupts	10
Summary of Input/Output Functions.....	10
Monitor Functions - Description.....	11
What happens at RESET?.....	14
Downloading and Running programs	14
Breakpoints - Things you should know!	15
Breakpoints, Single-Stepping and Interrupts	16
Appendix A - Downloadable File Formats	18
Appendix B - Error Messages.....	19
Appendix C - Input/Output Functions (Software Listings).....	20

Introduction

The MVS-31 system monitor is intended for use with the Quasar Electronics "SX-II", an 8031 8-bit micro controller based single board computer. This board may be used as a software development and debugging tool connected to a host computer. 8051 assembly language programs would be edited and assembled on the host system and then transferred to the SBC-31 board for execution and debugging.

MVS-31 Monitor/Debugger Features

- Display of internal and external data memory in hex and ascii.
- Modify internal and external data memory.
- Display all internal 8031 registers, including the Special Function Registers (SFRs).
- Modify all 8031 registers and SFRs individually.
- Disassemble code memory.
- Download files from a host PC in both Motorola S19 and Intel HEX formats.
- Runs code in REAL TIME.
- Program execution and debugging with jump to program, continue from breakpoint, single stepping, next "in-line" single stepping. Set watchpoints to display flags and variables.
- User accessible functions for various I/O routines such as input and output character, etc.

Getting Started

Refer to the "Getting Started - Running in 8031 Mode" section in the Southern Cross II Manual for details on configuring the hardware. Turn on the power when ready. The monitor uses automatic baud rate detection to configure the host port. Press Carriage Return (Enter↵). The following sign-on banner should be displayed.

```
*****
*           Quasar Electronics - SX-II           *
*****
* MVS-31 System Monitor      Version x.x   dd-mon-yy *
* Copyright (c) 1995   MicroValley Systems, Australia *
*****
```

Type "?" at the prompt to display a list of available commands. Invalid commands will be ignored and "?" displayed.

Monitor commands and conventions

MVS-31 is ready to accept keyboard commands whenever the ">" prompt is displayed. These commands consist of one or two characters followed by any hexadecimal parameters as required. Commands and parameters may be entered in upper or lower case.

Throughout this manual, the following conventions are used:

- The command typed by the user appears first.
- Parameters to be entered by the user are shown inside "<>" symbols (eg. <xxxx>). Entering a non-hex character during entry of hex parameters will cause the command to abort and a "?" displayed.
- Ascii characters are defined to be in the range \$20-\$7D
- All hexadecimal parameters are preceded by a "\$" symbol.

Command Set Summary

?	Display help screens
DA	DisAssemble code
DB	Display Breakpoints
DI	Display Internal memory in hex and ascii
DX	Display eXternal memory in hex and ascii
DR	Display Registers
DS	Display Special function registers (SFRs)
DT	Display Time and date
DW	Display Watchpoints
MI	Modify Internal memory
MX	Modify eXternal memory
ST	Set Time and date
<space>	Single step
N	Next "in-line" instruction single step
G	Go from current Program Counter value
J	Jump to user program
E	Enter breakpoint
C	Clear breakpoint
W	enter Watchpoint
K	Kill watchpoint
V	View watchpoint data
S0, S1	Load Motorola S19 format file
:	Load Intel HEX format file
A	Accumulator
B	B register
DPH	Data Pointer High byte
DPL	Data Pointer Low byte
DPTR	Data Pointer (16-bits)
IE	Interrupt Enable register
IP	Interrupt Priority register
P0	Port 0
P1	Port 1
P2	Port 2
P3	Port 3
PCON	Power Control register
PSW	Program Status Word
R0	Register 0
R1	Register 1
R2	Register 2
R3	Register 3
R4	Register 4
R5	Register 5
R6	Register 6
R7	Register 7
SBUF	Serial Port Buffer register
SCON	Serial Port Control register
SP	Stack Pointer
TCON	Timer/Counter Control register
TMOD	Timer/Counter Mode control register
TH0	Timer/Counter 0 High byte
TL0	Timer/Counter 0 Low byte
TH1	Timer/Counter 1 High byte
TL1	Timer/Counter 1 Low byte

Monitor Commands - Description

DA - DisAssemble program

Syntax: DA <xxxx>

Disassemble the binary code in program memory beginning at address <xxxx>. The display format is as follows.

```
1000: 020F2E    LJMP  0F2E
1003: EA MOV    A,R2
etc
```

Note that all numeric values are displayed in hexadecimal.

Instructions that contain a relative offset, such as JC, CJNE, DJNZ, AJMP, etc, have the actual destination address printed instead of the offset. Therefore the instruction

```
1004: B47005    CJNE  A,#70,100C
1007: xxxxxx
```

has the offset value (05) added to the address of the next instruction (1007) to form the destination address (100C).

One line of code is disassembled and printed and then output pauses. Press <CR> to exit the command or any other key to disassemble the next opcode.

DB - Display Breakpoints

Syntax: DB

Display the address of all currently set breakpoints.

DI - Display Internal data memory

Syntax: DI <xx>

Display the contents of internal data memory starting at address <xx> in hex and ascii, 16 bytes per line. Non ascii characters are displayed as a period (.)

This command displays the 16 bytes starting at address <xx> and pauses. Valid commands are:

```
<space> ..... next 16 bytes
<backspace> ..... previous 16 bytes
<CR> ..... exit
```

Address range is from \$00 to \$7F and is automatically wrapped around as needed.

DX - Display eXternal data memory

Syntax: DX <xxxx>

Display the contents of external data memory starting at address <xxxx> in hex and ascii, 16 bytes per line. Non ascii characters are displayed as a period (.).

Valid commands as per the "DI" command.

The address range is from \$0000 to \$FFFF and is automaticall wrapped around as needed.

DR - Display Registers

Syntax: DR

Display the current contents of the 8031 registers. The Program Status Word is displayed as individual bits. Two lines are printed as follows:

```
A  B  DPTR SP R0 R1 R2 R3 R4 R5 R6 R7 CAFrrOxP PC  Opcode Mnemonic
xx xx xxxx xx xx xx xx xx xx xx xx xxxxxxxx xxxx xxxxxxx xxxxxxxxxxxx
```

The first line is a heading and the second line is the contents of the corresponding register above it. A disassembled output of the opcode at the PC address is also displayed. The register bank displayed (R0-7) is determined by bits "rr" in the Program Status Word.

DS - Display Special function registers

Syntax: DS

Display the current contents of the 8031 Special Function Registers (SFRs). Two lines are printed as follows:

```
P0 P1 P2 P3 IE IP PCON SBUF SCON TCON TMOD TH0 TL0 TH1 TL1
xx xx xx xx xx xx xx  xx  xx  xx  xx  xx  xx xx xx  xx
```

DT - Display Time and date

Syntax: DT

Displays the current date and time. The time is displayed in 24 hour mode.

eg. Mon, 28-Aug-95 15:12:30

DW - Display Watchpoints

Syntax: DW

Display the address of all currently set watchpoints

MI - Modify Internal data memory

Syntax: MI <xx>

The contents of internal data memory at address <xx> are displayed in hex and new hex data may be entered. Address can be in the range \$00-7F. The following commands are valid:-

<space> next location
 <backspace> previous location
 <=> redisplay current location
 <CR> exit

Entering any other non-hex character will be followed by a "?" and the command aborted.

A "read-after-write" test is NOT performed when entering data.

The "=" command is useful for continually reading the contents of an I/O port or hardware register.

MX - Modify eXternal data memory

Syntax: MX <xxxx>

The contents of external data memory at address <xxxx> are displayed in hex and new hex data may be entered.

Refer to "MI" command for further details

ST - Set Time and date

Syntax: ST

Set the time and date on the clock/calendar chip. MVS-31 prints the following prompt and positions the cursor under the first character to be entered.

 MM/DD/YY DOW HH:MM:SS
 eg. 08/28/95 1 15:17:00

This sets the clock to Mon, August 8, 1995 3:17pm.

The characters "/" and ":" are automatically printed by the monitor and NOT entered by the user.

The clock runs in 24-hour mode. Therefore 1pm is entered as 13. No check is made for invalid values eg. Month (MM) = 17. Unpredictable results may occur if invalid values are entered.

Note: Day of Week (DOW) Sun = 0, Mon = 1, Tue = 2, Wed = 3, Thu = 4, Fri = 5, Sat = 6

<space> - Single step

Syntax: <space>

Execute the next instruction, stop and display registers and any watchpoint data. Control is transferred to the monitor. The program counter value shown is the address of the NEXT instruction to be executed. The opcode and mnemonic shown is that of the NEXT instruction to be executed.

Note: Calls by the user's program to the monitor's I/O routines are treated as a single instruction.

N - Next in-line instruction single stepping

Syntax: N

Operation of this command is similar to "space" type single stepping with the exception that "LCALL" and "ACALL" instructions cause the subroutine to be treated as a single instruction, ie executed without single stepping. Program execution stops at the next "in-line" instruction following the subroutine call.

G - Go from breakpoint

Syntax: G

Commence executing code from the current program counter address. Used mainly to continue program execution after a breakpoint. The code runs in "real time" until another breakpoint is encountered.

J - Jump to user program

Syntax: J <xxxx>

Jump to address <xxxx> and begin executing. A breakpoint at that address will cause the registers to be displayed and control returned to the monitor. The code runs in "real time" until a breakpoint is encountered.

E - Enter breakpoint

Syntax: E <xxxx>

Enter a breakpoint at address <xxxx>. A maximum of 8 breakpoints are allowed.

A number of checks are done when the breakpoint is first entered. These are:

Have the maximum number of breakpoints already been set?
Does a breakpoint already exist at that address?
Does a breakpoint already exist within 2 addresses either side of that address?.

The first two tests are self explanatory. The third test is to prevent one breakpoint from "overlapping" another. Refer to the section on "Breakpoints - Things you should know!" for further explanation.

C - Clear breakpoint

Syntax: C <xxxx>

Clear (delete) the breakpoint at address <xxxx>. Entering "X" or "x" instead of an address deletes ALL breakpoints currently set.

W - enter Watchpoint

Syntax: W <xx>

Enter a watchpoint at address <xx>. Watchpoints can only be set in internal data memory (address range \$00-7F). A maximum of 8 watchpoints are allowed.

A number of checks are done when entering the watchpoint, such as:

Have the maximum number of watchpoints already been set?
Does a watchpoint already exist at that address?

K - Kill watchpoint

Syntax: K <xx>

Kill (delete) the watchpoint at address <xx>. Address can be in the range \$00-7F. Entering "X" or "x" instead of an address deletes ALL watchpoints currently set.

V - View watchpoint data

Syntax: V

View (display) the contents at the watchpoint addresses. The watchpoint address is printed followed by a colon (:) followed by the data at that address.

S0 - Load a Motorola S19 format file**S1 - Load a Motorola S19 format file**

Syntax: S1

Loads a Motorola S19 formatted file into memory. Data transfer is via the host serial port. This command is normally terminated on receipt of an "S9" record. Clears any breakpoints already set.

There is no need to enter the "S1" command as this is part of the Motorola file format. Just send the file and the monitor will automatically detect it. Each "start of block" character ("S") is echoed back to the screen.

Load errors cause the command to abort immediately. Load errors are caused by either a non hex character received or by a checksum error. One of the following error messages is displayed whenever a load error occurs.

" Error: Bad hex character received ... Press <ESC> to exit "

" Error: Bad checksum ... Press <ESC> to exit "

No "read-after-write" testing occurs as the file is loaded. It is up to the user to ensure that the file is loaded into the correct memory locations.

: - Load an Intel HEX format file

Syntax: :

Load an Intel HEX formatted file into memory. Data transfer is via the host serial port. This command is normally terminated on receipt of an "EOF" record. Clears any breakpoints already set.

There is no need to enter the ":" command as this is part of the Intel file format. Just send the file and the monitor will automatically detect it. Each "start of block" character (":") is echoed back to the screen.

Refer to "S1" command for further details.

Register Modify commands

These commands allow the user to modify any of the internal Special Function Registers (SFRs) simply by entering its name. The monitor will respond with the address of the register and its current contents. Enter any new hex data as required or press <Enter> to exit.

Example:

```
>P0  
80: xx    (xx = current contents in hex)
```

Entering "DPTR" will display both "DPL and DPH" together and allow both to be modified by entering a single 16-bit value ("DPH" value first).

Note that the "R0-R7" registers accessed will depend on the current register bank selected in the "PSW" register.

User Accessible Input/Output Functions

The MVS-31 monitor provides a large number of I/O functions which can be called from user programs. These functions are accessed by executing an "LCALL <xxxx>" instruction from the user's program, where <xxxx> is the address of the routine required. The I/O function equates are supplied on disk in a file called "MVS31.EQU". Include this file at the start of your program to access these functions.

A number of 8031 registers are used to pass arguments between the I/O functions and the user program. Generally, 8-bit arguments are passed via the "A" accumulator and 16-bit arguments via the "DPTR" register. **Unless otherwise specified, all other registers are preserved.** Refer to the specific function for further details.

Some I/O routines require a number of bytes of user stack space. Refer to the software listings in Appendix C for further details.

All the "input hex" and "input decimal" calls exit with C = 0 if valid data is entered. If a non hex or decimal character is entered, then the routine exits with C = 1 and the character is returned in the A accumulator. All the input/output functions use the host serial port.

NOTE: Use a "LJMP NXTCMD" instruction to terminate the user's program. This method allows control to be passed back to the monitor when the program has finished

Input/Output Functions and Interrupts

All of these I/O functions use one or more 8031 registers, not only to pass arguments back and forth but also within the functions themselves. For example, inputting characters using "INCH" requires that, during execution of the function, the "DPTR" register hold the address of the host port, even though it is left unchanged once the function exits. If the "INCH" routine is interrupted whilst waiting for input and the "DPTR" register is altered before returning to the function then it will no longer be pointing to the host port. Any character returned will be garbage. Therefore, user interrupt routines should leave the "A" and "DPTR" registers unchanged. If they are to be used in the interrupt routine, push them on to the stack first and pop them off before exiting.

Summary of Input/Output Functions

NXTCMD	\$1FC0	Transfer control to MVS-31 monitor
ANYIN	\$1FC3	Check for character input
INCH	\$1FC6	Input character without input
INCHE	\$1FC9	Input character with echo
INHEX	\$1FCC	Input a hex digit
IN2HEX	\$1FCF	Input 2 hex digits
IN4HEX	\$1FD2	Input 4 hex digits
INDEC	\$1FD5	Input a decimal digit
BCD	\$1FD8	Input 2 decimal digits
OUTCH	\$1FDB	Output character
OUTHEX	\$1FDE	Output hex digit
OUT2H	\$1FE1	Output 2 hex digits
OUT2HS	\$1FE4	Output 2 hex digits followed by space
OUT4H	\$1FE7	Output 4 hex digits
OUT4HS	\$1FEA	Output 4 hex digits followed by space
OUTS	\$1FED	Output space
PDATA	\$1FF0	Output ascii string
PSTRING	\$1FF3	Output ascii string preceeded by CR, LF
PCRLF	\$1FF6	Output Carriage Return/Line Feed
RDTIME	\$1FF9	Read the time and date from the clock/calendar
WRTIME	\$1FFC	Write new time and date to clock/calendar

Monitor Functions - Description

NXTCMD	\$1FC0	Transfer control to the MVS-31 monitor Program control is transferred to the monitor.
ANYIN	\$1FC3	Check for character input C = 1 if a character received; C = 0 if no character received.
INCH	\$1FC6	Input character without echo The 8-bit character is returned in the A accumulator.
INCHE	\$1FC9	Input character with echo The character is returned in the A accumulator with the most significant bit (bit 7) clear. The character is also echoed to the terminal.
INHEX	\$1FCC	Input a hex digit The hex digit is returned in the right nibble of the A accumulator.
IN2HEX	\$1FCF	Input 2 hex digits The hex digits are returned in the A accumulator.
IN4HEX	\$1FD2	Input 4 hex digits The 4 hex digits are returned in the DPTR register. Any non-hex character will be returned in the A accumulator. If valid hex data is entered then the A accumulator is preserved.
INDEC	\$1FD5	Input a decimal digit The decimal digit is returned in the right nibble of the A accumulator.
BCD	\$1FD8	Input 2 decimal (BCD) digits The decimal digits are returned in the A accumulator
OUTCH	\$1FDB	Output character Output the character in the A accumulator to the terminal.

OUTHEX	\$1FDE	Output hex digit	The digit is contained in the right nibble of the A accumulator. The A accumulator is destroyed.
OUT2H	\$1FE1	Output 2 hex digits	Output the 2 hex digits (byte) in the A accumulator. The A accumulator is destroyed.
OUT2HS	\$1FE4	Output 2 hex digits followed by space	As per "OUT2H" but with a trailing space. The A accumulator is destroyed.
OUT4H	\$1FE7	Output 4 hex digits	Output the 4 hex digits (2 bytes) in the DPTR register. The A accumulator is destroyed.
OUT4HS	\$1FEA	Output 4 hex digits followed by space	As per "OUT4H" but with a trailing space. The A accumulator is destroyed.
OUTS	\$1FED	Output space	Output a space. The A accumulator is destroyed.
PDATA	\$1FF0	Output ascii string	Output the ascii string pointed to by the DPTR register. The string must be terminated by a NULL character (\$00). On exit, the DPTR register is left pointing to the NULL character. The A accumulator is destroyed.
PSTRNG	\$1FF3	Output string preceeded by CR,LF	As per "PDATA" except that carriage return/line feed preceeds the ascii string. The A accumulator is destroyed.
PCRLF	\$1FF6	Output Carriage Return/Line Feed	Output carriage return and line feed characters to the terminal. The A accumulator is destroyed.

RDTIME \$1FF9 Read the time and date from the clock/calendar

Returns 8 bytes of clock data into a memory buffer pointed to by the R0 register. **Internal memory must be used for this function.**

The data is in Binary Coded Decimal (BCD) format and is organised as follows:

Month, Date, Year, Day of Week, Hours, Minutes, Seconds, 1/100ths of seconds

The "Hours" byte is returned in 24-hour format (1pm = 13 hours).

The R0 register is unaffected. Register "A" is destroyed.

WRTIME \$1FFC Write new time and date to clock/calendar

On entry, register R0 points to an 8 byte memory buffer containing the new time and date information to be written into the clock/calendar chip. **Internal memory must be used for this function.**

The data should be in BCD format and organised as per the "RDTIME" function. The "hours" byte should be in 24-hour format. No check is made for valid data.

The R0 register is unaffected. Register "A" is destroyed.

What happens at RESET?

The monitor can detect the difference between a "power on reset" (POR) and pressing the RESET button. The POR sequence is as follows:

1. Set the monitor stack pointer (The monitor uses its own separate stack area).
2. Initialize the host serial port (automatic baud rate detection).
3. Print the signon banner.
4. Clear all breakpoint entries.
5. Initialize the user's interrupt vectors (at \$4000).
6. Initialize user's registers.
7. Print ">" prompt and wait for command.

Pressing the RESET button bypasses steps 4 and 5 above. This allows users to regain control by pressing "RESET" without losing any breakpoint information or changing any interrupt vectors already set by the user program.

Downloading and Running programs

Programs must first be assembled and then downloaded to the SBC-31 computer. The "host" port is used for downloading programs. The monitor will accept files in Motorola S19 or Intel HEX format. A description of these formats appears in Appendix A.

The SBC-31 has 16K of user program RAM starting at address \$4000. User programs should be ORG'd to start at this address, including the interrupt vectors. Except for reset and timer 2, all the interrupt vectors have been mapped into this area as follows:

Interrupt Source	8031 Interrupt Vectors	User Interrupt Vectors
External Interrupt 0	\$0003	\$4003
Timer 0 Interrupt	\$000B	\$400B
External Interrupt 1	\$0013	\$4013
Timer 1 Interrupt	\$001B	\$401B
Serial port interrupts	\$0023	\$4023

User Interrupt Vectors

The monitor contains an "LJMP" instruction at the 8031 interrupt vector address that transfers control to the user's interrupt vector. Therefore, user interrupt routines are delayed two instruction cycles by the "LJMP" instruction.

At power up, the user's interrupt vectors are initialized to jump to interrupt "trap" routines in the monitor. The trap routine prints a message identifying the interrupt source and the monitor then waits for a command.

User program RAM is mapped in both code and data space. Any instruction that accesses external data memory is also able to access the user program RAM.

NOTE: The monitor uses four bytes of user program RAM, beginning at address \$7FFC. Three bytes are used to assemble a program patch for accessing internal data RAM. The fourth byte is used to access an optional clock/calendar chip. User code must NOT reside in this area as it will be overwritten.

Breakpoints - Things you should know!

A short description of how breakpoints are implemented by the monitor is necessary in order that the user is aware of certain "anomalies" that can occur.

Breakpoints are not affected by pressing RESET but are automatically cleared after downloading a file.

Entering a breakpoint consists of replacing three bytes of the user's code, beginning at address <xxxx>, with an "LCALL" instruction into the monitor. The bytes are not replaced when the breakpoint is entered but rather when control is transferred to the user program via a "J, G, or N" command.

The user's code is intact in memory whenever the monitor program is in control.

Do not change any data at an address where a breakpoint is set (or the next two locations). The new data will be replaced with the original data when control is transferred back to the user's program.

A check is done when breakpoints are entered to test for "**overlapping**" breakpoints. The monitor will not allow a breakpoint to be entered if it is within two address locations either side of an existing breakpoint. This prevents one breakpoint from overwriting or "overlapping" another when they are inserted back into the user's program prior to running it.

The user's program is started with the "J" command. The program will run in **real time** until a breakpoint is encountered, at which point control is transferred to the monitor. The monitor restores the user's original code, displays the program registers (and any watchpoint data) and waits for a command. The program counter address shown is that of the NEXT instruction to be executed.

Press "G" to continue executing.

Wait on! What about the breakpoints?

When the breakpoint was encountered, the monitor restored the user's original code bytes ie. all the breakpoints were removed from the user's program. Before continuing to execute the user program, the monitor must put all the breakpoints back. However it cannot do this until AFTER it has executed the next instruction. If breakpoints were restored immediately, the next instruction would never be executed, as it would be a breakpoint. To overcome this problem, the monitor automatically executes a single step first. The monitor regains control after the single step, restores all breakpoints and then transfers control back to the user's program.

The user's code always runs in real time between breakpoints except for that first instruction.

Unfortunately, there is an exception to this rule!

Remember that a breakpoint replaces three bytes of user code.

This means that if a breakpoint is set at a one or two byte instruction, the following instruction(s) will also be affected. After executing the initial single step the monitor checks whether the next instruction to be executed is within two address locations of a breakpoint. If so, the monitor cannot replace breakpoints and continue as this would corrupt the next instruction. Therefore the monitor executes **another** single step. The process is repeated until the return address is at least three bytes away from the breakpoint address.

This does not normally present a problem as the user's program will be running in real time again within one or two instructions of a breakpoint. However, this will depend on the next instruction to be executed. An example will help to explain.

Consider the following program segment.

```

...
MOV   R6,#10                ;SET TIME DELAY REQUIRED
LOOP  CJNE  R6,#0,LOOP      ;WAIT UNTIL INTERRUPT COUNTER = 0
...

```

Example 1

Register "R6" is decremented within a timer interrupt routine. The software waits for "R6" to reach zero before continuing. A simple time delay using continuous timer interrupts!

Assume that a breakpoint is set at the "MOV" instruction (a two byte instruction). The program runs to the breakpoint, restores the user's code and stops. The user presses "G" to continue. The monitor executes a single step. The next instruction, "CJNE", is within two bytes of the breakpoint at the "MOV" instruction so the monitor executes another single step. However the program will not step past "LOOP" until register "R6" reaches zero. Therefore the monitor will continuously single-step the "CJNE" instruction. The program is not running in real time! The user's program will only run in real time once it has passed the "CJNE" instruction.

Setting breakpoints at one and two byte instructions can also have other consequences. Once again, an example will help.

```

AAAA  ...
      JC   BBBB                ;JUMP IF CARRY CLEAR
      ...
      ...
      CLR  A
      CLR  C
BBBB  MOV  P0,#55
      ...

```

Example 2

The two instructions before label "BBBB" are both single byte instructions. Setting a breakpoint at either of these instructions will also affect the "MOV" instruction. During execution it is possible that the program will bypass the breakpoint and jump directly to label "BBBB" (as at "AAAA"). However the "MOV" instruction has been altered by the breakpoint and the program could crash.

Most programmers collect subroutines and position them at the end of their code (I certainly do!). Setting a breakpoint within two bytes of the end of a subroutine would also affect the start of the following subroutine. That subroutine will not run correctly when called and could crash as well.

As a general rule, do not set breakpoints within two bytes prior to the destination of a "jump" instruction or the start of a subroutine.

Breakpoints, Single-Stepping and Interrupts

The single-stepping function in the monitor uses "Timer 2" of the 8032/52 processor. The timer is set to generate an interrupt during execution of the next instruction. The instruction will be completed and the interrupt will then transfer control back to the monitor. Some instructions, such as "RETI" and those accessing the "IE" and "IP" registers, block this interrupt until at least one more instruction has been executed. This could result in several instructions being executed before the monitor regains control.

The monitor disables all interrupts (clears the "EA" bit) whilst it is running. Therefore any interrupts occurring while the monitor is waiting for a command will be ignored. The monitor enables all interrupts (sets the "EA" bit) before transferring control back to the user's program via the "J, G, N or <space>" commands. If the user's program needs to stop interrupts it should do so on an individual basis and not by using the global "EA" bit.

In order for single-stepping to function correctly, the monitor initializes the interrupt priority of timer 2 to be higher than all the other interrupts (the "PT2" bit in the "IP" register is set and all others are cleared). Timer 2 interrupts are enabled by setting the "ET2" bit in the "IE" register. The "PT2" and "ET2" bits are always set by the monitor when single-stepping. Setting another interrupt source to the same priority level as timer 2 could affect the proper operation of the single-step function.

Example 3:

We want to single step through a timer interrupt routine. The timer interrupts are occurring at regular intervals, say 100uS. We set a breakpoint inside the interrupt routine and then start the program. The interrupt occurs and the program stops at the breakpoint. The monitor is waiting for a command. Meanwhile the user's timer is generating many more interrupts but they are ignored. When control is passed back to the user's program with a <space> command there will be two interrupts pending, one from timer 2 and one from the user's interrupt timer. If the interrupt timer has the same priority level as timer 2 it will get precedence and the single step will be ignored. The user's interrupt routine will interrupt itself and the user's stack will grow.

Wherever possible the interrupt priorities should be left unchanged. In this case all interrupts will be ignored when single-stepping.

As mentioned in the previous example, single-stepping through an interrupt routine is possible by first setting a breakpoint within the routine and then running the program. When the interrupt occurs, the program will hit the breakpoint and stop. All monitor commands are now available, including single-stepping. Single-stepping through an interrupt routine will ignore any further interrupts. However, pressing "G" will allow further interrupts to be processed. Should another interrupt be pending when "G" is pressed then it is possible that the interrupt routine will itself be interrupted before it can finish, causing the stack to overflow eventually. Of course the program will stop at the breakpoint again.

Interrupts will be processed when the user's program is running between breakpoints.

This is also true even if pressing "G" causes the program to continuously single step, as in example 1. Even though the program is not running in real time, interrupts will still be processed in between single steps.

Appendix A - Downloadable File Formats

Motorola S19 format

The Motorola S19 format breaks up a file into separate blocks or records. Each record is further divided into fields containing information about that record.

All fields contain ASCII characters to represent hexadecimal values. For example, the single hex byte "\$5A" is represented as two characters, ascii "5" and ascii "A".

The following table defines each field within a record.

Field	Description
Record Type	Two ASCII characters identifying the start and type of record. ASCII S1 - Data Record ASCII S9 - EOF Record
Byte Count	Two ASCII characters representing the number of bytes to follow, including the Address, Data and Checksum fields. An EOF record has a byte count of three.
Load Address	Four ASCII characters representing the address at which to start loading this record (high byte first). This field contains four ASCII zeros in an EOF record.
Data	Two ASCII characters representing each byte of data. There are no data bytes in an EOF record.
Checksum	Two ASCII characters representing the 1's complement of the sum of the byte count, load address and data fields.

Intel HEX format

The Intel HEX format is similar to the Motorola S19 format in that the file is also broken up into records and fields. All hexadecimal values are represented as ASCII characters.

The following table defines each field within a record.

Field	Description
Record Mark	An ASCII colon (:) indicates the start of a record.
Byte Count	Two ASCII characters representing the number of data bytes in this record. An EOF record has two ASCII zeros in this field.
Load Address	Four ASCII characters representing the address at which to start loading this record (high byte first). This field contains four ASCII zeros in an EOF record.
Record Type	Two ASCII characters identifying the type of record. 00 = Data Records 01 = EOF Record
Data	Two ASCII characters representing each byte of data. There are no data bytes in an EOF record.
Checksum	Two ASCII characters representing the 2's complement of the sum of the byte count, load address, record type and data fields.

Appendix B - Error Messages

Error: No breakpoint at this address

The breakpoint routine in the monitor has been called but a breakpoint does not exist at that address. The user's program is probably corrupt and should be downloaded again.

Error: Bad hex character received ... Press <ESC> to exit

An invalid hex character was received while downloading a file. Valid hex characters are "0-9, A-F" (upper or lower case). Abort the file transmission before pressing the <ESC> key. Check that the file is not corrupted and download again if all OK.

Error: Bad checksum ... Press <ESC> to exit

When downloading files, the monitor calculates a running checksum as each record is received. This checksum is compared to the checksum received at the end of the record. If they are not the same, this error message is printed. Abort the file transmission before pressing the <ESC> key. Check that the file is not corrupted and download again if all OK.

Error: Breakpoint table full

The maximum number of breakpoints have already been entered. Delete one or more breakpoints so that new breakpoints can be entered.

Error: Breakpoint already exists

A breakpoint has already been set at that address.

Error: Overlapping breakpoint

The user attempted to enter a breakpoint within two address locations of an existing breakpoint.

Error: No breakpoint there

The user tried to clear (delete) a breakpoint that does not exist

Error: Watchpoint table full

The maximum number of watchpoints have already been entered. Delete one or more watchpoints so that new watchpoints can be entered.

Error: Watchpoint already exists

A watchpoint has already been set at that address.

Error: No watchpoint there

The user tried to clear (delete) a watchpoint that does not exist

Appendix C - Input/Output Functions (Software Listings)

```

SPI          EQU    $F000                ;SERIAL PERIPHERAL INTERFACE (SCC2691)

*****
* SPI (SCC2691) REGISTER EQUATES
*****

;
MR1          EQU    SPI+0                ;MODE REGISTER 1
MR2          EQU    SPI+0                ;MODE REGISTER 2
SR           EQU    SPI+1                ;STATUS REGISTER          (READ)
CSR          EQU    SPI+1                ;CLOCK SELECT REGISTER    (WRITE)
CMR          EQU    SPI+2                ;COMMAND REGISTER        (WRITE)
RHR          EQU    SPI+3                ;RECEIVE HOLDING REGISTER (READ)
THR          EQU    SPI+3                ;TRANSMIT HOLDING REGISTER (WRITE)
ACR          EQU    SPI+4                ;AUXILIARY CONTROL REGISTER (WRITE)
ISR          EQU    SPI+5                ;INTERRUPT STATUS REGISTER (READ)
IMR          EQU    SPI+5                ;INTERRUPT MASK REGISTER  (WRITE)
CTU          EQU    SPI+6                ;COUNTER/TIMER UPPER     (READ)
CTUR         EQU    SPI+6                ;COUNTER/TIMER UPPER REGISTER (WRITE)
CTL          EQU    SPI+7                ;COUNTER/TIMER LOWER     (READ)
CTLR         EQU    SPI+7                ;COUNTER/TIMER LOWER REGISTER (WRITE)

*****
* PRINT ASCII STRING TO HOST PORT
* PRECEDED BY CR/LF
*****
;
; "DPTR" POINTS TO START OF STRING.
; STRING MUST BE TERMINATED BY "0".
;
; EXIT CONDITIONS AS PER "PDATA"
;
PSTRING      LCALL PCRLF                ;OUTPUT CR/LF FIRST

*****
* OUTPUT ASCII STRING TO HOST PORT
*****
;
; "DPTR" POINTS TO START OF STRING.
; STRING MUST BE TERMINATED BY "0".
;
; ON EXIT:      "A" DESTROYED
;              : "DPTR" POINTS TO STRING TERMINATOR ("0")
;
PDATA        CLR    A                    ;OFFSET=0
              MOVC  A,@A+DPTR            ;GET CHARACTER
              JZ    PDATAX               ;EXIT IF TERMINATOR ("0")
              LCALL OUTCH                 ;ELSE PRINT CHARACTER
              INC   DPTR                  ;POINT TO NEXT CHARACTER
              SJMP  PDATA                  ;CONTINUE WITH REST OF STRING
PDATAX        RET                          ;END OF STRING - FINISHED

```

```

*****
* OUTPUT CR/LF TO HOST PORT
*****
;
; ON EXIT:      "A" DESTROYED
;
PCRLF      MOV   A,#CR           ;SEND CARRIAGE RETURN
           LCALL OUTCH
           MOV   A,#LF           ;FOLLOWED BY LINE FEED

*****
* OUTPUT CHARACTER IN "A" TO HOST PORT
*****
;
; NO REGISTERS AFFECTED
;
OUTCH      PUSH  DPH             ;SAVE "DPTR"
           PUSH  DPL
           PUSH  ACC             ;SAVE CHARACTER
           MOV   DPTR,#SR        ;POINT TO UART STATUS REGISTER
OUTCH1     MOVX  A,@DPTR         ;GET UART STATUS
           JNB   A.2,OUTCH1      ;WAIT FOR TRANSMITTER READY
           MOV   DPTR,#THR       ;POINT TO TRANSMIT REGISTER
           POP   ACC             ;RETRIEVE CHARACTER
           MOVX  @DPTR,A         ;SEND CHARACTER
           POP   DPL             ;RESTORE "DPTR"
           POP   DPH
           RET

*****
* OUTPUT 4 HEX DIGITS IN "DPTR"
* FOLLOWED BY A SPACE
*****
;
; EXIT CONDITIONS AS PER "OUT4H"
;
OUT4HS     LCALL OUT4H           ;OUTPUT THE 4 DIGITS
           SJMP  OUTS

*****
* OUTPUT 2 HEX DIGITS IN "A"
* FOLLOWED BY A SPACE
*****
;
; EXIT CONDITIONS AS PER "OUT2H"
;
OUT2HS     LCALL OUT2H           ;OUTPUT BYTE

*****
* OUTPUT A SPACE
*****
;
; ON EXIT: "A" DESTROYED
;
OUTS       MOV   A,#$20
           SJMP  OUTCH

*****

```

```

* OUTPUT 4 HEX DIGITS IN "DPTR"
*****
;
; ON EXIT:      "A" DESTROYED
;              "DPTR" PRESERVED
;
OUT4H          MOV    A,DPH                ;GET FIRST BYTE
               LCALL OUT2H              ;PRINT IT
               MOV    A,DPL                ;GET SECOND BYTE

*****
* OUTPUT 2 HEX DIGITS IN "A"
*****
;
; ON EXIT:      "A" DESTROYED
;
OUT2H          PUSH  ACC                  ;SAVE BYTE
               SWAP  A                    ;READY TO OUTPUT FIRST DIGIT
               LCALL OUTHEX
               POP   ACC                  ;RETRIEVE BYTE

*****
* OUTPUT RIGHT NIBBLE IN "A"
*****
;
; ON EXIT:      "A" DESTROYED
;
OUTHEX         ANL   A,#$0F                ;STRIP OFF LEFT NIBBLE
               ADD   A,#$30                ;CONVERT TO ASCII
               CJNE  A,#$3A,OUTHEX1       ;WAS DIGIT "0-9" ?
OUTHEX1        JC    OUTHEX2              ;JUMP IF SO
               ADD   A,#$07                ;ELSE MUST BE LETTER "A-F"
OUTHEX2        SJMP OUTCH                  ;OUTPUT DIGIT

*****
* CHECK IF ANY INPUT FROM HOST PORT
*****
;
; ON EXIT:      C=0 IF NONE
;              C=1 IF SO
;
ANYIN          PUSH  DPH                  ;SAVE "DPTR"
               PUSH  DPL
               PUSH  ACC                  ;SAVE "A"
               MOV   DPTR,#SR             ;POINT TO UART STATUS REGISTER
               MOVX  A,@DPTR              ;GET UART STATUS
               CLR   C                      ;ASSUME NO INPUT
               JNB  A.0,ANYIN1            ;JUMP IF NO INPUT
               SETB C                      ;ELSE SET CARRY FLAG (INPUT!)
ANYIN1         POP   ACC
               POP   DPL
               POP   DPH
               RET

```

```

*****
* INPUT 8-BIT CHARACTER FROM HOST PORT INTO "A"
*****
;
INCH          PUSH  DPH          ;SAVE "DPTR"
              PUSH  DPL
              MOV   DPTR,#SR     ;POINT TO UART STATUS REGISTER
INCH1        MOVX  A,@DPTR      ;GET UART STATUS
              JNB   A.0,INCH1    ;WAIT FOR INPUT
              MOV   DPTR,#RHR    ;POINT TO RECEIVE REGISTER
              MOVX  A,@DPTR      ;GET CHARACTER
              POP   DPL
              POP   DPH
              RET

*****
* INPUT 7-BIT CHARACTER WITH ECHO INTO "A"
*****
;
INCHE        LCALL INCH         ;GET 8-BIT CHARACTER
              ANL   A,#$7F      ;STRIP OF HIGH BIT
              LJMP OUTCH        ;AND OUTPUT IT!

*****
* GET HEX ADDRESS (4 DIGITS) INTO "DPTR"
*****
;
; EXIT CONDITIONS AS PER "INHEX"
;
IN4HEX       LCALL IN2HEX       ;GET FIRST BYTE (2 DIGITS)
              JC    IN4HEXX     ;EXIT IF ERROR
              MOV   DPH,A        ;SAVE IT
              LCALL IN2HEX       ;GET SECOND BYTE
              JC    IN4HEXX     ;EXIT IF ERROR
              MOV   DPL,A        ;SAVE IT
              CLR   C
IN4HEXX      RET

*****
* GET HEX BYTE (2 DIGITS) INTO "A"
*****
;
; EXIT CONDITIONS AS PER "INHEX"
;
IN2HEX       PUSH  B            ;SAVE "B" REGISTER
              LCALL INHEX       ;GET FIRST HEX DIGIT
              JC    IN2HEXX     ;EXIT IF NOT HEX
              SWAP A            ;PUT DIGIT INTO HIGH NIBBLE
              MOV   B,A         ;SAVE IT
              LCALL INHEX       ;GET SECOND DIGIT
              JC    IN2HEXX     ;EXIT IF NOT HEX
              ADD   A,B         ;ADD IN FIRST DIGIT
              CLR   C
IN2HEXX      POP   B            ;RESTORE "B" REGISTER
              RET

```

```

*****
* GET HEX CHARACTER (1 DIGIT) INTO "A"
*****
;
; ON EXIT:      C=1 IF CHARACTER IS NOT HEX DIGIT (0-9, A-F)
;               C=0 OTHERWISE
;
INHEX          LCALL INCHE          ;GET ASCII CHARACTER
               CJNE  A,#$30,INHEX1  ;IS IT < "0" ?
INHEX1         JC    NOTHEX         ;JUMP IF SO - ERROR!
               CJNE  A,#$3A,INHEX2  ;IS IT > "9" ?
INHEX2         JNC  INHEX3         ;JUMP IF SO
               CLR   C
               SUBB  A,#$30         ;CONVERT ASCII TO HEX
               RET
INHEX3         ANL   A,#$DF         ;CONVERT TO UPPER CASE
               CJNE  A,#$41,INHEX4  ;IS CHARACTER < "A" ?
INHEX4         JC    NOTHEX         ;JUMP IF SO - ERROR!
               CJNE  A,#$47,INHEX5  ;IS IT > "F" ?
INHEX5         JNC  NOTHEX         ;JUMP IF SO - ERROR!
               CLR   C
               SUBB  A,#$37         ;CONVERT ASCII TO HEX
               RET

; HERE IF INVALID HEX OR DECIMAL DIGIT
;
NOTHEX         EQU    *
NOTDEC         SETB  C              ;SET CARRY BIT - ERROR
               RET

*****
* GET DECIMAL CHARACTER (1 DIGIT) INTO "A"
*****
;
; ON EXIT:      C=1 IF CHARACTER IS NOT DECIMAL DIGIT (0-9)
;               C=0 OTHERWISE
;
INDEC          LCALL INCHE          ;GET ASCII CHARACTER
               CJNE  A,#$30,INDEC1  ;IS IT < "0" ?
INDEC1JC      NOTDEC              ;JUMP IF SO - ERROR!
               CJNE  A,#$3A,INDEC2  ;IS IT > "9" ?
INDEC2JNC    NOTDEC              ;JUMP IF SO
               CLR   C
               SUBB  A,#$30         ;CONVERT ASCII TO DECIMAL
               RET

*****
* GET BCD BYTE (2 DECIMAL DIGITS) INTO "A"
*****
;
; EXIT CONDITIONS AS PER "INDEC"
;
BCD           PUSH  B              ;SAVE "B" REGISTER
               LCALL INDEC         ;GET FIRST DECIMAL DIGIT
               JC    BCDX          ;EXIT IF ERROR
               SWAP A              ;MOVE TO HIGH NIBBLE
               MOV  B,A            ;SAVE INTO "B"
               LCALL INDEC         ;GET SECOND DECIMAL DIGIT

```

```

                JC      BCDX
                ADD    A,B          ;COMBINE WITH FIRST DIGIT
                CLR   C
BCDX           POP    B          ;RESTORE "B" REGISTER
                RET

```

```
*****
```

```
* WRITE TIME TO CLOCK/CALENDAR CHIP
```

```
*****
```

```

;
; ON ENTRY:  "R0" = ADDRESS OF 8 BYTE BLOCK CONTAINING NEW TIME DATA.
;            DATA BLOCK MUST BE IN "INTERNAL" RAM AREA.
;            DATA FORMAT = YY, MM, DD, DOW, HRS, MINS, SECS, SECS/100
; ON EXIT:   "A" DESTROYED
;            NO OTHER REGISTERS AFFECTED
;
;
WRTIME        MOV    A,R0          ;MOVE POINTER TO END OF DATA AREA
                ADD    A,#7
                MOV    R0,A
WRTIME1       LCALL  RTC_OPEN      ;ACCESS CLOCK CHIP
                MOV    A,@R0       ;GET "SECS/100"
                LCALL  WRCLOCK     ;WRITE IT
                DEC    R0
                MOV    A,@R0       ;GET "SECONDS"
                LCALL  WRCLOCK
                DEC    R0
                MOV    A,@R0       ;GET "MINUTES"
                LCALL  WRCLOCK
                DEC    R0
                MOV    A,@R0       ;GET "HOURS"
                ANL   A,#%01111111 ;SET FOR 24-HOUR MODE
                LCALL  WRCLOCK
                DEC    R0
                MOV    A,@R0       ;GET "DAY-OF-WEEK"
                ORL   A,#%00010000 ;IGNORE "RESET" PIN
                ANL   A,#%11011111 ;TURN ON "OSCILLATOR"
                LCALL  WRCLOCK
                DEC    R0
                MOV    A,@R0       ;GET "DATE"
                LCALL  WRCLOCK
                DEC    R0
                MOV    A,@R0       ;GET "MONTH"
                LCALL  WRCLOCK
                DEC    R0
                MOV    A,@R0       ;GET "YEAR"
                LCALL  WRCLOCK
                CLR   C
                RET

```

```

*****
* READ TIME FROM CLOCK/CALENDAR CHIP
*****
;
; ON ENTRY:  "R0" = ADDRESS OF 8 BYTE BLOCK WHERE CLOCK DATA WILL BE
;            RETURNED.
;            DATA BLOCK MUST BE IN "INTERNAL" RAM AREA.
;            DATA FORMAT = YY, MM, DD, DOW, HRS, MINS, SECS, SECS/100
; ON EXIT:   "A" DESTROYED
;            NO OTHER REGISTERS AFFECTED
;
RDTIME      MOV    A,R0                ;MOVE POINTER TO END OF DATA AREA
            ADD    A,#7
            MOV    R0,A
            LCALL  RTC_OPEN           ;ACCESS CLOCK CHIP
            LCALL  RDCLOCK           ;GET "SECS/100"
            MOV    @R0,A             ;SAVE IT
            DEC    R0
            LCALL  RDCLOCK           ;GET "SECS"
            MOV    @R0,A             ;SAVE IT
            DEC    R0
            LCALL  RDCLOCK           ;GET "MINS"
            MOV    @R0,A             ;SAVE IT
            DEC    R0
            LCALL  RDCLOCK           ;GET "HRS"
            ANL    A,#%01111111     ;ALWAYS 24-HOUR MODE
            MOV    @R0,A             ;SAVE IT
            DEC    R0
            LCALL  RDCLOCK           ;GET "DOW"
            ANL    A,#$07           ;MASK OFF ALL BUT "DOW" BITS
            MOV    @R0,A             ;SAVE IT
            DEC    R0
            LCALL  RDCLOCK           ;GET "DATE"
            MOV    @R0,A             ;SAVE IT
            DEC    R0
            LCALL  RDCLOCK           ;GET "MONTH"
            MOV    @R0,A             ;SAVE IT
            DEC    R0
            LCALL  RDCLOCK           ;GET "YEAR"
            MOV    @R0,A ;SAVE IT
            RET
;
*****
* OPEN ACCESS TO THE CLOCK/CALENDAR CHIP
*****
;
; FIRST EXECUTE 72 READS TO ENSURE THAT THE RTC IS CLOSED,
; THEN WRITE THE 64-BIT RECOGNITION PATTERN.
;
; THE HEX RECOGNITION PATTERN IS C5,3A,A3,5C,C5,3A,A3,5C (LSB TO MSB)
;
; ON EXIT, "A" DESTROYED
;
RTC_OPEN    PUSH   DPH
            PUSH   DPL
            PUSH   B
            MOV    DPTR,#RTCADDR     ;POINT TO CLOCK/CALENDAR CHIP
            MOV    B,#72             ;SET UP FOR 72 READS
RTC_OP1     MOVX   A,@DPTR           ;READ DATA BIT

```

```

                DJNZ B,RTC_OP1           ;LOOP UNTIL DONE
                MOV  B,#2                 ;SET UP TO SEND 2 BLOCKS OF "C5,3A,A3,5C"
RTC_OP2        MOV  A,#$C5               ;SEND "C5"
                LCALL WRCLOCK
                MOV  A,#$3A              ;SEND "3A"
                LCALL WRCLOCK
                MOV  A,#$A3              ;SEND "A3"
                LCALL WRCLOCK
                MOV  A,#$5C              ;SEND "5C"
                LCALL WRCLOCK
                DJNZ B,RTC_OP2           ;LOOP UNTIL BOTH BLOCKS SENT
                POP  B
                POP  DPL
                POP  DPH
                RET

```

* WRITE A BYTE TO THE CLOCK/CALENDAR CHIP

```

;
; ON ENTRY:  "A" = BYTE TO WRITE
; ON EXIT:   ALL REGISTERS PRESERVED
;
WRCLOCK       PUSH  DPH
               PUSH  DPL
               MOV   DPTR,#RTCADDR      ;POINT TO CLOCK/CALENDAR CHIP
               MOVX  @DPTR,A           ;WRITE BIT 0
               RR    A
               MOVX  @DPTR,A           ;WRITE BIT 1
               RR    A
               MOVX  @DPTR,A           ;WRITE BIT 2
               RR    A
               MOVX  @DPTR,A           ;WRITE BIT 3
               RR    A
               MOVX  @DPTR,A           ;WRITE BIT 4
               RR    A
               MOVX  @DPTR,A           ;WRITE BIT 5
               RR    A
               MOVX  @DPTR,A           ;WRITE BIT 6
               RR    A
               MOVX  @DPTR,A           ;WRITE BIT 7
               RR    A
               POP   DPL
               POP   DPH
               RET

```

* READ A BYTE FROM THE CLOCK/CALENDAR CHIP

```

;
; ON EXIT:    BYTE RETURNED IN "A"
;
RDCLOCK      PUSH   DPH
              PUSH   DPL
              PUSH   B
              MOV    DPTR,#RTCADDR    ;POINT TO CLOCK/CALENDAR CHIP
              MOV    B,#8             ;READ 8 BITS
RDCLOCK1     PUSH   ACC               ;SAVE ACCUMULATOR
              MOVX  A,@DPTR          ;READ DATA BIT
              RRC   A                 ;MOVE DATA BIT INTO CARRY
              POP   ACC               ;RETRIEVE ACCUMULATOR
              RRC   A                 ;MOVE DATA BIT INTO IT
              DJNZ  B,RDCLOCK1       ;LOOP UNTIL ALL BITS READ
              POP   B
              POP   DPL
              POP   DPH
              RET

```