GMX®

GMX 020BUG v2.7x

020Bug Debugging Package

User's Manual

January 1987

GMX 020BUG v2.7x

020Bug Debugging Package

User's Manual

First GMX Edition February 1986

GMX Inc.
1337 W. 37th Place
Chicago, IL 60609
312-927-5510 * TWX 910-221-4055

First Motorola Edition October 1984

GMX 020Bug Debugging Package Manual

Revision History

Revision A	02/04/86	First edition
Revision B	04/29/86	Second edition
Revision C	07/21/86	Third edition
Revision D	08/22/86	Fourth edition
Revision E	09/16/86	Fifth edition
Revision F	11/04/86	Sixth edition
Revision G	01/13/87	Seventh edition

TABLE OF CONTENTS

LIST OF TABLES

CHAPTER 1

GENERAL INFORMATION

1.1 DESCRIPTION OF 020Bug

The 020Bug package is a powerful evaluation and debugging tool for systems built around the GMX Micro-20 processor module. Facilities are available for loading and executing user programs under complete operator control for system evaluation. 020Bug includes commands for display and modification of memory, breakpoint capabilities, a powerful assembler/disassembler useful for patching programs, and a self-test on power up feature which verifies the integrity of the system. Various 020Bug routines that handle I/O, data conversion, and string functions are available to user programs through the TRAP #15 handler.

020Bug consists of three parts; (1) a command-driven user-interactive software debugger, described in Chapter 2 and hereafter referred to as the "debugger", (2) a command-driven diagnostic package for the GMX Micro-20 hardware, described in Appendix D and hereafter referred to as the "diagnostics", and (3) a user interface which accepts commands from the system console terminal.

When using 020Bug the user will either operate out of the debugger directory or out of the diagnostic directory. If the user is in the debugger directory then the debugger prompt, "020Bug>", will be displayed and the user will have all of the debugger commands at his disposal. If the user is in the diagnostic directory then the diagnostic prompt, "M20Diag>", will be displayed and the user will have all of the diagnostic commands at his disposal as well as all of the debugger commands. The user may switch between directories by using the "SD" command, described in section 3.23 or may examine the commands in the particular directory that he is currently in by using the "HE" command, described in section 3.12.

Since 020Bug is command-driven, it performs its various operations in response to user commands entered at the keyboard. Figure 1-1 illustrates the flow of control in 020Bug. When a command is entered, 020Bug will execute the command and the prompt will reappear. However, if a command is entered which causes execution of user target code (i.e., "GO") then control may or may not return to 020Bug, depending on the outcome of the user program.

Those users who have used one or more of Motorola's other debugging packages (i.e., MACSbug, VERSAbug, TENbug, etc.) will find 020Bug very similar. There are two noticeable differences. Many of the commands are more flexible and powerful. Also, the debugger in general is more "user-friendly", with more detailed error messages and an expanded on-line help facility.

1.2 HOW TO USE THIS GUIDE

If the user has never used a debugging package before, then he should read all of Chapter 1 before attempting to use 020Bug. This will give an idea of 020Bug's structure and capabilities.

For a question about syntax or operation of a particular 020Bug command, the user may turn to the entry for that particular command in the section describing the command set (Chapter 3).

Some debugger commands take advantage of the built-in one-line assembler/disassembler. The command descriptions in Chapter 3 assume that the user already understands how the assembler/disassembler works. See the assembler/disassembler description in Chapter 4 for details on its use.

1.3 INSTALLATION AND STARTUP

Procedures for installing the GMX Micro-20 and setting it up for operation are described in the "GMX Micro-20 Hardware Setup Manual".

1.4 SWITCHES

Positions 1 and 2 of the DIP switch bank S1 on the GMX Micro-20 board determine what happens upon power-up or reset after the confidence test is completed. If switch S1-1 is OFF, then the GMX Micro-20 will begin execution of the alternate ROM program, fetching the start address and initial stack value as described in Appendix A, "Alternate ROM Programs".

If switch S1-1 is ON, then switch S1-2 is tested. If S1-2 is OFF then execution continues in 020BUG: the user will see the 020BUG startup message and prompt, and can enter 020BUG commands.

If S1-1 is ON, and S1-2 is ON, then 020BUG will enter a special mode. In this mode, 020BUG does not prompt the user for a command, but immediately begins execution of the self-test diagnostics in a special mode. See sections 4.9 (Loop Continuous) and 4.11 (Self Test Led) in Appendix D.

1.5 RESTARTING THE SYSTEM

There are three ways for the user to initialize the system to a known state. Each has characteristics which make it more appropriate than another in certain situations.

1.5.1 Reset

Depressing and releasing the RESET button connected to the RESET input of the GMX Micro-20 will initiate a system reset. The processor's program counter and stack pointer are loaded from the first two longwords of the ROM, and execution begins at the address so specified. During the reset routine a total system initialization takes place, as if the GMX Micro-20 had just been powered up. All static variables are restored to their default states. The Breakpoint table is cleared. The offset registers are cleared. The target registers are invalidated. Input and output character queues are cleared. Serial ports 0, 1, and 2 are reconfigured to their default state. The other on-board devices (port 3, the PI/T, the FDC, and the SASI), and any devices connected to the I/O expansion bus are reset, but not reinitialized. The Time-of-day clock is not affected.

Reset must be used if the processor ever halts (as evidenced by the GMX Micro-20's halt light glowing), for example after a double bus fault, or if the 020Bug environment is ever lost (vector table is destroyed, etc).

1.5.2 RS command

This command causes the processor to load the program counter and stack pointer from the first two longwords of the ROM, just as if the RESET button had been pressed. The RESET routine includes a RESET instruction, which causes the processor to send out a RESET signal to the rest of the system, so performing the RS command also causes hard reset.

1.5.3 Abort

Abort is invoked by pressing and releasing the ABORT button connected to the GMX Micro-20 ABORT input. Whenever Abort is invoked, a "snapshot" of the processor state is captured and stored in the target registers. For this reason Abort is most appropriate when terminating a user program that is being debugged. Abort should be used to regain control if the program gets caught in a loop, etc. The target PC, stack pointers, etc will help to pinpoint the malfunction. Abort generates a level seven interrupt (non-maskable). The target registers, reflecting the machine state at the time the ABORT button was pushed, will be displayed to the screen. Control will be returned to the debugger.

1.5.4 Break

A "Break" is generated by pressing and releasing the BREAK key on the terminal keyboard. Break does not take a snapshot of the machine state nor does it display the target registers. The user may want to terminate a debugger command before its completion, for example, the display of a large block of memory. Break allows the user to terminate the command without overwriting the contents of the target registers, as would be done if Abort were used. Break does not interrupt program operation in any way; it is only detected by polling during serial I/O. A program which does no serial I/O cannot be interrupted by Break.

1.6 MEMORY REQUIREMENTS

The program portion of 020Bug is approximately 64K bytes of code. The EPROM sockets on the GMX Micro-20 are mapped at locations $00800000 to $0083FFFF. The first 64K bytes of this space are reserved for 020BUG. 020Bug requires a minimum of 16K bytes of read/write memory at $00000000 to operate. The first 8K bytes are used for 020Bug stack and static variable space and the next 8K bytes is reserved as user space. Whenever the GMX Micro-20 is reset the target program counter is initialized to the address corresponding to the beginning of the user space and the target stack pointers are initialized to addresses within the user space, with the target ISP set to the top of the user space. The limits for memory testing are set at Reset to $00020000 and $01FFFFC.

1.7 TERMINAL INPUT/OUTPUT CONTROL

When entering a command at the prompt the following control codes may be entered for limited command line editing. (Note: The presence of the upward caret "^" before a character indicates that the Control or "CTRL" key must be held down while striking the character key).

^X (cancel line) - The cursor is backspaced to the beginning of the line. If the terminal port is configured with the hardcopy or TTY option (see PF command), then a CR/LF and prompt is issued also.

^H (backspace) - The cursor is moved back one position. The character at the new position is erased. If the hardcopy option is selected, a "/" is typed along with the deleted character.

<DEL> (delete or rubout) - Same as ^H

^D (redisplay) - The entire command line as entered so far is redisplayed on the following line.

When any program or command is sending output to the 020BUG console, the XON/XOFF characters which are in effect for the terminal port may be entered to control the output, if the XON/XOFF protocol is enabled (default). These characters are initialized to ^S and ^Q respectively by 020Bug but may be changed by the user using the PF command. In the initialized (default) mode operation is as follows:

^S (wait) - Console output is halted.

^Q (resume) - Console output is resumed.

1.8 DIAGNOSTIC FACILITIES

Included in the 020Bug packages are a complete set of hardware diagnostics intended for testing and troubleshooting of the GMX Micro-20. In order to use the diagnostics the user must be in the diagnostic directory of 020Bug. In the debugger directory, the user can switch to the diagnostic directory by entering the debugger command "SD" for "switch directories". The diagnostic prompt ("M20Diag>") should appear. See Appendix D for complete descriptions of the diagnostic routines availables and instructions on how to invoke them.

CHAPTER 2

USING THE 020Bug DEBUGGER

2.1 ENTERING DEBUGGER COMMAND LINES

As mentioned previously, 020Bug is command-driven and performs its various operations in response to user commands entered at the keyboard. When the debugger prompt ("020Bug>") appears on the terminal screen then the debugger is ready to accept commands.

As the command line is entered it is stored in an internal buffer. Execution begins only after the carriage return is entered, thus allowing the user to correct entry errors if necessary using the control characters described in section 1.7.

When a command is entered the debugger will execute the command and the prompt will reappear. However, if the command entered causes execution of user target code, for example "GO", then control may or may not return to the debugger, depending on what the user program does. For example, if a breakpoint has been specified then control will return to the debugger when the breakpoint is encountered during execution of the user program. Alternately the user program could return control to the debugger by means of the TRAP #15 function ".RETURN" (described in section 5.2.18). For more about this, see the description in sections 3.8, 3.10, and 3.11 for the GO, GD, and GT commands.

In general, a debugger command is made up of the following parts:

1) The command identifier (i.e., "MD" for the memory display command).

2) A port number if the command is set up to work with more than one port.

3) At least one intervening space before the first argument.

4) Any required arguments, as specified by command.

5) An option field, set off by a semicolon (;) to specify conditions other than the default conditions of the command.

The commands are shown using a modified Backus-Naur form syntax. The metasymbols used are:

< > The angular brackets enclose a symbol, known as a syntactic variable, that is replaced in a command line by one of a class of symbols it represents.

[ ] Square brackets enclose a symbol that is optional.

| This symbol indicates that a choice is to be made. One of several symbols, separated by this symbol, should be selected.

/ The slash indicates that one or more of the symbols separated by this symbol can be selected.

{ } These brackets enclose an optional symbol that may occur zero or more times.

2.1.1 Syntactic Variables

The following syntactic variables will be encountered in the command descriptions which follow. In addition, other syntactic variables may be used and will be defined in the particular command description in which they occur.

<DEL> - Delimiter; either a comma or a space.
<EXP> - Expression (described in detail in section 2.1.1.1).
<ADDR> - Address (described in detail in section 2.1.1.2).
<COUNT> - Count; the syntax is the same as for <EXP>.
<RANGE> - A range of memory addresses which may be specified either by <ADDR> <DEL> <ADDR> or by <ADDR> : <COUNT>.
<TEXT> - An ASCII string of up to 255 characters, delimited at each end by the single quote mark (').

2.1.1.1 Expression as a Parameter

An expression can be one or more numeric values separated by the arithmetic operators plus (+) or minus (-). Numeric values may be expressed in either hexadecimal, decimal, octal or binary by immediately preceding them with the proper base identifier.

Base	Identifier	Examples
Hexadecimal	$	$FFFFFFFF
Decimal	&	&1974, &10-&4
Octal	@	@456
Binary	2	21000110

If no base identifier is specified then the numeric value is normally assumed to be hexadecimal. The only exceptions occur when using the one-line assembler/disassembler, which assumes a default of decimal, and when using the PF command, which also assumes a default of decimal.

Examples of valid expressions:

Expression	Result (in hexadecimal)
FF0011	FF0011
45+99	DE
&45+&99	90
@35+@67+@10	5C
210010110	96
210011110+21001	A7

The total value of the expression must be between 0 and $FFFFFFFF.

2.1.1.2 Address as a Parameter

Many commands use <ADDR> as a parameter. The syntax accepted by 020Bug is similar to the one accepted by the MC68020 one-line assembler. All control addressing modes are allowed except the PC-relative modes. An address + offset register mode is also provided.

2.1.1.2.1 Address Formats

Table 2-1 summarizes the address formats which are acceptable for address parameters in debugger command lines.

TABLE 2-1. FORMATS FOR DEBUGGER ADDRESS PARAMETERS

Format	Example	Description
N	140	Absolute address + contents of automatic offset register.
N+Rn	130+R5	Absolute address + contents of the specified offset register (not an assembler-accepted syntax).
(An)	(A1)	Address register indirect.
(d,An) or d(An)	(120,A1) 120(A1)	Address register indirect with displacement (two formats accepted).
(d,An,Xn) or d(An,Xn)	(&120,A1,D2) &120(A1,D2)	Address register indirect with index and displacement (two formats accepted).
([bd,An,Xn],od)	([C,A2,A3],&100)	Memory indirect pre-indexed.
([bd,An],Xn,od)	([12,A3],D2,&10)	Memory indirect post-indexed.

For the memory indirect modes, fields can be omitted, but two consecutive commas are not allowed. For example, two of many permutations are as follows:

([An],od)	([A1],4)
([bd])	([FC1E])

The following is not a valid mode:

([bd,,Xn])	([8,,D2])

Notes:
- N - Absolute address (any valid expression)
- An - Address register n
- Xn - Index register n (An or Dn)
- d - Displacement (any valid expression)
- bd - Base displacement (any valid expression)
- od - Outer displacement (any valid expression)
- n - Register number (0 to 7)
- Rn - Offset register n

2.1.1.2.2 Offset Registers

Eight pseudo-registers (R0-R7) called offset registers are used to simplify the debugging of relocatable and position independent modules. The listing files in these types of programs usually start at an address (normally 0) that is not the one in which they are loaded, so it is harder to correlate addresses in the listing with addresses in the loaded program. The offset registers solve this problem by taking into account this difference and forcing the display of addresses in a relative address+offset form, with the relative address portion matching the listing address. Address arguments required by 020Bug commands can also be entered in the relative address+offset format.

Example: The listing file of a relocatable module assembled with the MC68020 VERSAdos Resident Assembler is shown below:

```
 1
 2                                *
 3                                * MOVE STRING SUBROUTINE
 4                                *
 5  0 00000000 48E78080           MOVESTR  MOVEM.L  D0/A0,-(A7)
 6  0 00000004 4280                        CLR.L    D0
 7  0 00000006 1018                        MOVE.B   (A0)+,D0
 8  0 00000008 5340                        SUBQ.W   #1,D0
 9  0 0000000A 12D8                LOOP    MOVE.B   (A0)+,(A1)+
10  0 0000000C 51C8FFFC            MOVS    DBRA     D0,LOOP
11  0 00000010 4CDF0101                    MOVEM.L  (A7)+,D0/A0
12  0 00000014 4E75                        RTS
13
14                                         END

******  TOTAL ERRORS       0--
******  TOTAL WARNINGS     0--
```

The above program was loaded at address 0001327C. The disassembled code is shown next:

```
020Bug> MD 1327C;DI <CR>
0001327C 48E78080                MOVEM.L  D0/A0,-(A7)
00013280 4280                    CLR.L    D0
00013282 1018                    MOVE.B   (A0)+,D0
00013284 5340                    SUBQ.W   #1,D0
00013286 12D8                    MOVE.B   (A0)+,(A1)+
00013288 51C8FFFC                DBF      D0,$13286
0001328C 4CDF0101                MOVEM.L  (A7)+,D0/A0
00013290 4E75                    RTS
020Bug>
```

By using one of the offset registers, the disassembled code addresses can be made to match the listing file addresses as follows:

```
020Bug> OF R0 <CR> R0 =00000000?  1327C.<CR>
020Bug> MD 0+R0;DI <CR>
00000+R0 48E78080                MOVEM.L  D0/A0,-(A7)
00004+R0 4280                    CLR.L    D0
00006+R0 1018                    MOVE.B   (A0)+,D0
00008+R0 5340                    SUBQ.W   #1,D0
0000A+R0 12D8                    MOVE.B   (A0)+,(A1)+
0000C+R0 51C8FFFC                DBF      D0,$0000A+R0
00010+R0 4CDF0101                MOVEM.L  (A7)+,D0/A0
00014+R0 4E75                    RTS
020Bug>
```

For additional information about the offset registers, see the OF command description.

2.1.2 Port Numbers

Some 020Bug commands give the user the option of choosing the port which will be used to input or output. The valid port numbers which may be used for these commands are:

0 - GMX Micro-20 Terminal Port (DUART #1 channel A)
1 - GMX Micro-20 Host Port (DUART #1 channel B)
2 - GMX Micro-20 Printer Port (DUART #2 channel A)

2.1.3 Multiple commands on a line

A single line of 020Bug input may contain more than one 020BUG commands. No separator is required between command strings if the earlier command does not accept parameters (i.e., the "SD" command). In other cases, a command string can be terminated with a "!" character, which is treated like an end of line by the command parsing routine. Command strings of any complexity can be placed on the same line provided they are separated by "!" chracters.

2.2 ENTERING AND DEBUGGING PROGRAMS

There are two ways to enter a user program into system memory for execution. One way is to create the program using the MM (Memory Modify) command with the assembler/disassembler option. The program is entered by the user one source line at a time. After each source line is entered, it is assembled and the object code is loaded to memory. Refer to Chapter 4 for complete details of the 020Bug Assembler/Disassembler.

The other way to enter a program is to download an object file from a host system, for example, an EXORmacs. The program must be in S-Record format (described in Appendix C) and may have been assembled or compiled on the host system. Alternately the program may have been previously created using the 020Bug MM command as outlined above and stored to the host using the DU (Dump) command. A communication link must exist between the host system and the GMX Micro-20's port 1 (see hardware configuration details in the GMX Micro-20 Hardware Setup Manual). The file is downloaded from the host into GMX Micro-20 memory with the debugger's LO command.

Once the object code has been loaded into memory, the user can set breakpoints if desired and run the code or trace through it.

2.3 CALLING SYSTEM UTILITIES FROM USER PROGRAMS

A convenient way of doing character input/output and many other useful operations has been provided so that the user does not have to write these routines into the target code. The user has access to various 020Bug routines via the MC68020 TRAP #15 instruction. Refer to Chapter 5 for details on the various TRAP #15 utilities available and how to invoke them from within a user program.

2.4 PRESERVING THE DEBUGGER OPERATING ENVIRONMENT

This section explains how to avoid contaminating the operating environment of the debugger. 020Bug uses certain of the GMX Micro-20's on-board resources to contain temporary variables, exception vectors, etc. If the user disturbs resources which 020Bug depends on, then the debugger may function unreliably or not at all.

2.4.1 020Bug Vector Table and Workspace

As described in section 1.6, "Memory Requirements", 020Bug needs 8K bytes of read/write memory to operate and also allocates another 8K bytes as user space for a total of 16K bytes allocated at $00000000. On power-up/reset, the exception vector table is built there, and the MC68020 Vector Base Register (VBR) is made to point to the start of the table. Next, 020Bug reserves space for static variables and initializes these static variables to predefined default values. After the static variables, 020Bug allocates space for the system stack and then initializes the system stack pointer to the top of this area.

The user must be extremely careful not to use the above-mentioned memory areas for other purposes. If, for example, a user program inadvertently wrote over the static variable area containing the serial communication parameters, these parameters would be lost, resulting in a loss of communication with the system console terminal. If a user program corrupts the system stack then an incorrect value may be loaded into the processor's program counter, causing a system crash.

2.4.2 Maintaining a User Vector Table

If a user program is written to do exception processing, it must write the address of its exception handling routine(s) into the vectors at the proper offset from the address pointed to by the VBR. Often it is desirable to switch from running the user program to running the 020Bug debugger and back again. In these cases it is necessary to avoid conflicts between exception vectors used by the user program and exception vectors used by the debugger.

The exception vectors used by the debugger are shown in Table 2-2. Of these vectors, only the TRAP #15 vector is absolutely necessary for the debugger to operate. Any time that the debugger code is entered the vector at offset $BC in the original 020Bug vector table MUST contain the address of the 020Bug TRAP #15 handler. This is because the debugger uses the system call functions internally to do console I/O and other functions. Other vectors in Table 2-2 are required if the user desires to use the associated debugger facilities (breakpoints, trace mode, etc.).

TABLE 2-2. EXCEPTION VECTORS USED BY 020Bug

Offset from Vector Base	Exception	020Bug Facility
$10	Illegal Instruction	Breakpoints (Used by GO, GN, GT)
$24	Trace	T, TC, TT
$BC	TRAP #15	System calls (See Chapter 5)
$100	Level 7 Interrupt	ABORT pushbotton

When the debugger handles one of the exceptions listed in Table 2-2, the target stack pointer is left pointing past the bottom of the exception stack frame created; that is, it reflects the system stack pointer values just before the exception occurred. In this way, the operation of the debugger facility (through an exception) is transparent to the user.

Example: Trace one instruction using the debugger.

```
020Bug> RD <CR>
PC  =00002000 SR  =2700=TR:OFF S. 7 .....
USP =00003830 MSP =00003C18 ISP*=00004000 VBR =00000000
SFC =0=XX     DFC =0=XX     CACR=0=..     CAAR=00000000
D0  =00000000 D1  =00000000 D2  =00000000 D3  =00000000
D4  =00000000 D5  =00000000 D6  =00000000 D7  =00000000
A0  =00000000 A1  =00000000 A2  =00000000 A3  =00000000
A4  =00000000 A5  =00000000 A6  =00000000 A7  =00004000
00002000 203900100000                MOVE.L  ($100000).L,D0
020Bug> T <CR>
PC  =00002006 SR  =2700=TR:OFF S. 7 .....e
USP =00003830 MSP =00003C18 ISP*=00004000 VBR =00000000
SFC =0=XX     DFC =0=XX     CACR=0=..     CAAR=00000000
D0  =12345678 D1  =00000000 D2  =00000000 D3  =00000000
D4  =00000000 D5  =00000000 D6  =00000000 D7  =00000000
A0  =00000000 A1  =00000000 A2  =00000000 A3  =00000000
A4  =00000000 A5  =00000000 A6  =00000000 A7  =00004000
00002006 D280                        ADD.L  D0,D1
020Bug>
```

Notice that the value of the target stack register has not changed even though a trace exception has taken place. The user program may either share the exception vector table used by 020Bug or it may create a separate exception vector table of its own. The two following sections detail these two methods.

2.4.3.1 Sharing 020Bug's Vector Table

The user program may share the 020Bug vector table by simply writing its own exception vectors into the vector table. 020Bug uses only a few of the many MC68020 vector locations to operate. These vectors are listed in Table 2-2.

The user program may even overwrite some of the debugger's vectors if the user does not need those particular debugger functions. Care must be taken, however, not to overwrite the TRAP #15 vector because this exception is used internally by the debugger.

The beginning address of the 020Bug vector table is loaded into the target-state VBR at power-up and reset. The user can find out what this beginning address is by resetting the GMX Micro-20 using RESET, then displaying the target state registers using the RD command. The value displayed for the VBR will be the 020Bug vector table start address.

2.4.3.2 Creating a Separate Vector Table

A user program may create a separate vector table in memory to contain its exception vectors. Then the user program must change the value of the VBR to point at the new vector table. In order to use the debugger facilities the user can copy the proper vectors from the 020Bug vector table into the corresponding vector locations in the user vector table.

As mentioned in section 2.4.2, 020Bug saves the address of its vector table and workspace such that it may be recovered if the user changes the MC68020's VBR contents and then re-enters 020Bug. When the debugger code is entered, 020Bug will automatically change the VBR to point at its original vector table. The VBR value set up by the user will be saved in the target-state VBR.

The vector for the 020Bug generalized exception handler (described in detail in section 2.4.3.3) may be copied from offset $08 (Bus Error vector) in 020Bug's vector table to all locations in the user's vector table where a separate exception handler is not used. This will provide diagnostic support in the event that the user program is stopped by an unexpected exception. The generalized exception handler gives a formatted display of MC68020 registers and identifies the type of the exception.

The following is an example of a user routine which builds a separate vector table and then moves the VBR to point at it:

```
*
****  BUILDX - Build exception vector table  ****
*
BUILDX  MOVEC.L  VBR,A0              Get copy of 020Bug VBR.
        LEA      $10000,A1           New vectors at $10000.
        MOVE.L   $8(A0),D0           Copy generalized exception vector.
        MOVE.W   $3FC,D1             Load count (all vectors).
LOOP    MOVE.L   D0,(A1,D1)          Store generalized exception vector.
        SUBQ.W   #4,D1
        BNE.B    LOOP                Initialize entire vectortable.
        MOVE.L   $10(A0),$10(A1)     Copy breakpoints vector.
        MOVE.L   $24(A0),$24(A1)     Copy trace vector.
        MOVE.L   $BC(A0),$BC(A1)     Copy system call vector.
        LEA.L    COPROCC(PC),A2      Get user exception vector.
        MOVE.L   A2,$2C(A1)          Install as F-Line handler.
        MOVEC.L  A1,VBR              Change VBR to new table.
        RTS
        END
```

It may turn out that the user program uses one or more of the exception vectors that are required for debugger operation. Debugger facilities may still be used, however, if the user's exception handler can determine when to handle the exception itself and when to pass the exception to the debugger.

When an exception occurs which the user wants to pass on to the debugger (ABORT, for example) the user's exception handler must read the vector offset from the format word of the exception stack frame. This offset is added to the address of the 020Bug vector table (which the user program saved), yielding the address of the 020Bug exception vector. The user program then jumps to the address stored at this vector location, which is the address of the 020Bug exception handler.

The user program must make sure that there is an exception stack frame in the stack and that it is exactly the same as the processor would have created it for the particular exception before jumping to the address of the exception handler. Below is an example of a user exception handler which can pass an exception along to the debugger:

```
*
****  EXCEPT - Exception handler  ****
*
EXCEPT  SUBQ.L    #4,A7                 Reserve a longword in the stack.
        LINK      A6,#0                 New vectors at $10000.
        MOVEM.L   A0-A5/D0-D7,-(SP)
        MOVE.L    BUGVBR,A0             Pass exception to debugger; Get VBR.
        MOVE.W    14(A6),D0             Get the vector offset from stack frame.
        AND.W     #$0FFF,D0             Mask off the format information.
        MOVE.L    (A0,D0.W),4(A6)       Store address of debugger exc handler.
        MOVEM.L   (SP)+,A0-A6/D0-D7
        RTS                             Put address of exc handler in PC and go.
```

2.4.3.3 020Bug Generalized Exception Handler

020Bug has a generalized exception handler which it uses to handle all of the exceptions not listed in Table 2-2. For all these exceptions, the target stack pointer is left pointing to the top of the exception stack frame created. In this way, if an unexpected exception occurs, the user is presented with the exception stack frame to help determine the cause of the exception. The following example illustrates this:

Example: Bus error at address $F00000. It is assumed for this example that an access of memory location $F00000 will initiate Bus Error exception processing.

```
020Bug> RD <CR>
PC  =00002000 SR  =2700=TR:OFF S. 7 .....
USP =00003830 MSP =00003C18 ISP*=00004000 VBR =00000000
SFC =0=XX     DFC =0=XX     CACR=0=..     CAAR=00000000
D0  =00000000 D1  =00000000 D2  =00000000 D3  =00000000
D4  =00000000 D5  =00000000 D6  =00000000 D7  =00000000
A0  =00000000 A1  =00000000 A2  =00000000 A3  =00000000
A4  =00000000 A5  =00000000 A6  =00000000 A7  =00004000
00002000 203900F00000                MOVE.L  ($F00000).L,D0
020Bug> T <CR>
```

```
Exception: Long Bus Error
Format/Vector=B008
SSW=0145 Fault Addr.=00F00000 Data In=00000000 Data Out=00002006
PC  =00002000 SR  =A700=TR:ALL S. 7 .....
USP =00003830 MSP =00003C18 ISP*=00003FA4 VBR =00000000
SFC =0=XX     DFC =0=XX     CACR=0=..     CAAR=00000000
D0  =00000000 D1  =00000000 D2  =00000000 D3  =00000000
D4  =00000000 D5  =00000000 D6  =00000000 D7  =00000000
A0  =00000000 A1  =00000000 A2  =00000000 A3  =00000000
A4  =00000000 A5  =00000000 A6  =00000000 A7  =00003FA4
00002000 203900F00000                MOVE.L  ($F00000).L,D0
020Bug>
```

Notice that the target stack pointer is different. The target stack pointer now points to the last value of the exception stack frame that was stacked. The exception stack frame may now be examined using the MD command:

```
020Bug> MD (A7):&44 <CR>
00003FA4 A700  0000  2000  B008   3E2C  0145  0000  0027      '... .0.>,.E...'
00003FB4 0F00  0000  0F00  0000   0000  1BCC  2039  0000      .p...p.....L 9..
00003FC4 0000  200A  0000  2008   0000  2006  0000  0000      .. ... ... .....
00003FD4 00F0  0000  100F  0487   0000  A700  4003  0000      .p........'.@...
00003FE4 0000  7FFF  0000  0000   C010  0000  0000  4000      ........@.....@.
00003FF4 0000  0000  FFF8  086C                               .....x.l
020Bug>
```

CHAPTER 3

THE 020Bug DEBUGGER COMMAND SET

3.1 INTRODUCTION This chapter contains descriptions of each of the debugger commands. It also provides one or more examples of each command. Table 3-1 summarizes the 020Bug debugger commands, grouped by type.

TABLE 3-1. DEBUGGER COMMANDS BY TYPE

Each of the individual commands is described in the following pages. The command's syntax is shown using the symbols explained in section 2.1.

In the examples shown, all user input is underlined. This is done for clarity in understanding the examples (to distinguish between characters input by the user and characters output by 020Bug). No underline is typed in actual input. The symbol <CR> represents the carriage return key on the user's terminal keyboard. Whenever this symbol appears it means that a carriage return was entered by the user.

BF <RANGE><DEL><data> [; B:W:L]

where:

<data> is an expression parameter

options:

B - Byte
W - Word
L - Longword

The BF command fills the specified range of memory with the specified data pattern. The data entered by the user is right-justified in either a byte, word or longword field (as specified by the option selected). The default field length is W (word).

If the user-entered data does not fit into the data field size then leading bits are truncated to make it fit. If truncation occurs then a message will be printed stating the data pattern which was actually written.

If the range is specified using a count then the count is assumed to be in terms of the data size.

If the upper address of the range is not on the correct boundary for an integer multiple of the data to be stored then data is stored to the last boundary before the upper address. No address outside of the specified range will ever be disturbed in any case. The "Effective address" messages displayed by the command will show exactly where data was stored.

Example 1: (Assume memory from $20000 to $2002F is clear)

```
020Bug> BF 20000,2001F 4E71 <CR>
Effective address: 00020000
Effective address: 0002001F
020Bug> MD 20000:30;B <CR>
00020000 4E 71 4E 71 4E 17 4E 71   4E 71 4E 71 4E 71 4E 71    NqNqNqNqNqNqNqNq
00020010 4E 71 4E 71 4E 17 4E 71   4E 71 4E 71 4E 71 4E 71    NqNqNqNqNqNqNqNq
00020020 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00    ................
```

Since no option was specified, the length of the data field defaulted to word.

Example 2: (Assume memory from $20000 to $2002F is clear)

```
020Bug> BF 20000:10 4E71 ;B <CR>
Effective address: 00020000
Effective count  : &16
Data = $71
020Bug> MD 20000:30;B <CR>
00020000 71 71 71 71 71 17 71 71   71 71 71 71 71 71 71 71    qqqqqqqqqqqqqqqq
00020010 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00    ................
00020020 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00    ................
```

The specified data did not fit into the specified data field size. The data was truncated and the "Data = " message was output.

Example 3: (Assume memory from $20000 to $2002F is clear)

```
020Bug> BF 20000,20006 12345678 ; L <CR>
Effective address: 00020000
Effective address: 00020003
020Bug> MD 20000:30;B <CR>
00020000 12 34 56 78 00 00 00 00   00 00 00 00 00 00 00 00     .4Vx............
00020010 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00     ................
00020020 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00     ................
```

The longword pattern would not fit evenly in the given range. Only one longword was written and the "Effective address" messages reflect the fact that data was not written all the way up to the specified address.

3.3 BLOCK OF MEMORY MOVE BM

BM <RANGE><DEL><ADDR> [; B:W:L]

options:

B - Byte
W - Word
L - Longword

The BM command copies the contents of the memory addresses defined by <RANGE> to another place in memory, beginning at <ADDR>.

The option field is only allowed when <RANGE> was specified using a count. In this case the B, W, or L defines the size of data that the count is referring to. For example a count of 4 with an option of L would mean to move 4 longwords (or 16 bytes) to the new location. If an option field is specified without a count in the range an error results.

Example 1: (Assume memory from 20000 to 2000F is clear)

```
020Bug> MD 21000:20;B <CR>
00021000 54 48 20 53 20 49 53 20   41 20 54 45 53 54 21 21     THIS IS A TEST!!
00021010 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00     ................

020Bug> BM 21000 2100F 20000 <CR>
Effective address: 00021000
Effective address: 0002100F
Effective address: 00020000

020Bug> MD 20000:20;B <CR>
00021000 54 48 49 53 20 49 53 20   41 20 54 45 53 54 21 21     THIS IS A TEST!!
00021010 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00     ................
020Bug>
```

Example 2: This utility is very useful for patching assembly code in memory. Suppose the user had a short program in memory at address 20000...

```
020Bug> MD 20000 2000A;DI <CR>
00020000 D480                      ADD.L    D0,D2
00020002 E2A2                      ASR.L    D1,D2
00020004 2602                      MOVE.L   D2,D3
00020006 4E4F                      TRAP     #15
00020008 0021                      DC.W     $21
0002000A 4E71                      NOP
```

Now suppose the user would like to insert a NOP between the ADD.L instruction and the ASR.L instruction. The user should Block Move the object code down two bytes to make room for the NOP.

```
020Bug> BM 20002 2000B 20004 <CR>
Effective address: 00020002
Effective address: 0002000B
Effective address: 00020004
```

```
020Bug> MD 20000 2000C;DI <CR>
00020000 D480                           ADD.L    D0,D2
00020002 E2A2                           ASR.L    D1,D2
00020004 E2A2                           ASR.L    D1,D2
00020006 2602                           MOVE.L   D2,D3
00020008 4E4F                           TRAP     #15
0002000A 0021                           DC.W     $21
0002000C 4E71                           NOP
```

Now the user needs only to enter the NOP at address 20002.

```
020Bug> MM 20002;DI <CR>
00020002 E2A2                           ASR.L    D1,D2 ?  NOP <CR>
00020002 4E71                           NOP
00020004 E2A2                           ASR.L    D1,D2 ?
020Bug>

020Bug> MD 20000 2000C;DI <CR>
00020000 D480                           ADD.L    D0,D2
00020002 4E71                           NOP
00020004 E2A2                           ASR.L    D1,D2
00020006 2602                           MOVE.L   D2,D3
00020008 4E4F                           TRAP     #15
0002000A 0021                           DC.W     $21
0002000C 4E71                           NOP
020Bug>
```

3.4 BREAKPOINT INSERT/DELETE

BR
NOBR

```
BR   [<ADDR>[:<COUNT>]]
NOBR [<ADDR>]
```

The BR command allows the user to set a target code instruction address as a "breakpoint address" for debugging purposes. If during target code execution a breakpoint with 0 count is found, the target code state is saved in the target registers and control is returned back to 020Bug. This allows the user to see the actual state of the processor at selected instructions in the code.

Up to eight breakpoints can be defined. The breakpoints are kept in a table which is displayed each time either BR or NOBR are used. If an address is specified with the BR command that address is added to the breakpoint table. The count field specifies how many times the instruction at the breakpoint address must be fetched before a breakpoint is taken. The count, if greater than zero, is decremented with each fetch. Every time that a breakpoint with zero count is found, a breakpoint handler routine prints the CPU state on the screen and control is returned to 020Bug.

NOBR is used for deleting breakpoints from the breakpoint table. If an address is specified then that address will be removed from the breakpoint table. If NOBR <CR> is entered then all entries will be deleted from the breakpoint table and the empty table will be displayed.

Example:

```
020Bug> BR 14000,14200  14700:&12 <CR>          Set Some Breakpoints
BREAKPOINTS
00014000            00014200
00012700:C
020Bug> NOBR 14200 <CR>                         Delete One Breakpoint
BREAKPOINTS
00014000            00012700:C
020Bug> NOBR <CR>                               Delete All Breakpoints
BREAKPOINTS
020Bug>
```

3.5 BLOCK OF MEMORY SEARCH BS

BS <RANGE> <DEL> <TEXT> [;B:W:L]

or

BS <RANGE> <DEL> <data> <DEL> [<mask>] [;B:W:L,N]

The block search command searches the specified range of memory for a match with a user-entered data pattern. This command has two modes, as described below.

Mode 1 - LITERAL STRING SEARCH -- In this mode a search is carried out for the ASCII equivalent of the literal string entered by the user. This mode is assumed if the single quote (') indicating the beginning of a <TEXT> field is encountered following <RANGE>. The size as specified in the option field tells whether the count field of <RANGE> refers to bytes, words, or longwords. If <RANGE> is not specified using a count then no options are allowed. If a match is found, then the address of the first byte of the match is output.

Mode 2 - DATA SEARCH -- In this mode a data pattern is entered by the user as part of the command line and a size is either entered by the user in the option field or is assumed (the assumption is word). The size entered in the option field also dictates whether the count field in <RANGE> refers to bytes, words, or longwords. The following actions occur during a data search:

1) The user-entered data pattern is right-justified and leading bits are truncated or leading zeroes are added as necessary to make the data pattern the specified size.

2) A compare is made with successive bytes, words, or longwords (depending on the size in effect) within the range for a match with the user-entered data. Comparison is made only on those bits at bit positions corresponding to a "1" in the mask. If no mask is specified then a default mask of all one's is used (all bits will be compared). The size of the mask is taken to be the same size as the data.

3) If the "N" ("Non-aligned") option has been selected then the data is searched for on a byte-by-byte basis, rather than by words or longwords regardless of the size of <data>. This is useful if a word (or longword) pattern is being searched for, but is not expected to lie on a word (or longword) boundary.

4) If a match is found then the address of the first byte of the match is output along with the memory contents. If a mask was in use then the actual data at the memory location is displayed, rather than the data with the mask applied.

For both modes, information on matches is output to the screen in a four-column format. If more than 24 lines of matches are found then output is inhibited to prevent the first match from rolling off of the screen. A message is printed at the bottom of the screen indicating that there is more to display. To resume output the user should simply depress any character key. To cancel the output and exit the command the user should press the break key.

If a match is found with a series of bytes of memory whose beginning is within the range but whose end is outside of the range then that match will be output and a message will be output stating that the last match does not lie entirely within the range. The user may search non-contiguous memory with this command without causing a Bus Error.

Examples: (Assume the following data is in memory).

```
00030000 00 00 00 45 72 72 6F 72   20 53 74 61 74 75 73 3D     ...Error Status=
00030010 34 46 2F 2F 43 6F 6E 66   69 67 54 61 62 6C 65 53     4F//ConfigTableS
00030020 74 61 72 74 3A 00 00 00   00 00 00 00 00 00 00 00     tart:...........
```

```
020Bug> BS 30000 3002F 'Task Status'_<CR>
Effective address: 00030000
Effective address: 0003002F
-not found-

020Bug> BS 30000 3002F 'Error Status'_<CR>
Effective address: 00030000
Effective address: 0003002F
00030003

020Bug> BS 30000 3001F 'ConfigTableStart'_<CR>
Effective address: 00030000
Effective address: 0003001F
00030014 -last match extends over range boundary-

020Bug> BS 30000:30 't'_;_<CR>
Effective count  : &48
0003000A   0003000C   00030020   00030023

020Bug> BS 30000:18,2F2F <CR>
Effective address: 00030000
Effective count  : &24
00030012:2F2F

020Bug> bs 30000,3002F 3d34 <CR>
Effective address: 00030000
Effective address: 0003002F
-not found-

020Bug> bs 30000,3002F 3d34 ;n <CR>
Effective address: 00030000
Effective address: 0003002F
0003000F:3D34

020Bug> BS 30000:30 60,F0 ;B <CR>
Effective address: 00030000
Effective count  : &48
00030006:6F   0003000B:61   000300015:6F   00030016:6E

00030017:66   00030018:69   000300019:67   0003001B:61

0003001C:62   0003001D:6C   00030001E:65   00030021:61
```

3.6 DATA CONVERSION DC

DC <EXP> ¦ <ADDR>

The DC command is used to simplify an expression into a single numeric value. This equivalent value is displayed in its hexadecimal and decimal representation. If the numeric value could be interpreted as a signed negative number (i.e., if the most significant bit of the 32-bit internal representation of the number is set) then both the signed and unsigned interpretations are displayed.

DC can also be used to obtain the equivalent effective address of an MC68020 addressing mode.

Examples:

```
020Bug> DC 10 <CR>
          00000010 = $10 = &16

020Bug> DC &10-&20 <CR>
SIGNED   : FFFFFFF6 = -$A = -&10
UNSIGNED: FFFFFFF6 = $FFFFFFF6 = &4294967286

020Bug> DC 123+&345+@67+%1100001 <CR>
          00000314 = $314 = &788
```

The subsequent examples assume A0=00030000 and the following data resides in memory:

```
00030000 11111111   22222222   33333333   44444444    ....""""3333DDDD

020Bug> DC (A0) <CR>
          00030000 = $30000 = &196608

020Bug> DC ([A0]) <CR>
          11111111 = $11111111 = &286331153

020Bug> DC (4,A0) <CR>
          00030004 = $30004 = &196612

020Bug> DC ([4,A0]) <CR>
          22222222 = $22222222 = &572662306
```

3.7 DUMP S-RECORDS DU

DU[<port>]<DEL><RANGE><DEL>[<TEXT><DEL>][<ADDR>][;B:W:L]

The DU command outputs data from memory in the form of Motorola S-Records to a port specified by the user. If port is not specified then the S-Records are sent to the host port.

The option field is allowed only if a count was entered as part of the range and defines the units of the count (bytes, words or longwords).

The optional <TEXT> field is for text that will be incorporated into the header (S0) record of the block of records that will be dumped.

The optional <ADDR> field is to allow the user to enter an entry address for code contained in the block of records. This address is incorporated into the address field of the block's termination record. If no entry address is entered then the address field of the termination record will consist of zeroes. The termination record will be an S7, S8, or S9 record, depending on the address entered. See Appendix C for additional information on S-Records.

Example 1: Dump memory from $20000 to $2002F to port 1.

```
020Bug> DU 20000 2002F <CR>
Effective address: 00020000
Effective address: 0002002F
020Bug>
```

Example 2: Dump 10 bytes of memory beginning at $30000 to the terminal screen (port 0).

```
020Bug> DU0 30000:&10 <CR>
Effective address: 00030000
Effective count  : &10
S0030000FC
S20E0300002602544553546608 4E4F7B
S9030000FC
020Bug>
```

Example 3: Dump memory from $20000 to $2002F to host (port 1). Specify a file name of "TEST" in the header record and specify an entry point of $2000A.

```
020Bug> DU 20000 2002F 'TEST' 2000A <CR>
Effective address: 00020000
Effective address: 0002002F
020Bug>
```

The following example shows how to upload S-Records to a host computer (in this case an EXORmacs running the VERSAdos operating system), storing them in the file "FILE1.MX" which the user will create with the VERSAdos utility UPLOADS.

```
020Bug> TM <CR>                  ( Go into transparent mode to establish    )
Escape character: $01=^A         (communication with the EXORmacs.           )
<BREAK>                          ( Press BREAK key to get VERSAdos login     )
                                 ( prompt.                                   )
   "
```

```
   " (login)                        ( User must log onto VERSAdos and enter the   )
   "                                ( catalog where FILE1.MX will reside.         )
   "
= UPLOADS FILE1 <CR>                ( At VERSAdos prompt invoke the UPLOADS       )
                                    ( utility and tell it to create a file        )
                                    ( named "FILE1" for the S-Records that will   )
                                    ( be uploaded.                                )
```

The UPLOADS utility will at this point display some messages like the following:

```
                          UPLOAD "S" RECORDS
                             Version x.y
                       Copyrighted by MOTOROLA, INC.
volume=xxxx
 catlg=xxxx
  file=FILE
   ext=MX

UPLOADS Allocating new file

Ready for "S" records,...

= <_^A>                             ( When the VERSAdos prompt returns enter the  )
020Bug>                             ( escape character to return to 020Bug.       )
```

Now enter the command for 020Bug to dump the S-Records to the port.

```
020Bug> DU 20000 2000F 'FILE1' <CR>
Effective address: 00020000
Effective address: 0002000F
020Bug>

020Bug> TM <CR>                     ( Go into transparent mode again.             )
Escape character: $01=^A
QUIT <CR>                           ( Tell UPLOADS to quit looking for records.   )
```

The UPLOADS utility will now display some more messages like this:

```
                          UPLOAD "S" RECORDS
                             Version x.y
                       Copyrighted by MOTOROLA, INC.
volume=xxxx
 catlg=xxxx
  file=FILE
   ext=MX

*STATUS*  No error since start of program

Upload of S-Records complete.

= OFF <CR>                          ( The VERSAdos prompt should return.          )
                                    ( Log off of the EXORmacs.                    )
<_^A>                               ( Enter the escape character to return to     )
020Bug>                             ( 020Bug.                                     )
```

3.8 GO DIRECT (IGNORE BREAKPOINTS) GD

GD [<ADDR>]

GD is used to start target code execution. If an address is specified, it is placed in the target PC. Execution starts at the target PC address. As opposed to GO, breakpoints are not inserted.

Once execution of target code has begun, control may be returned to 020Bug by various conditions:

1) The user pressed ABORT or RESET.
2) An unexpected exception occurred.
3) By execution of the .RETURN TRAP #15 function.

Example: (The following program resides at $10000)

```
020Bug> MD 10000;DI <CR>
00010000 2200                    MOVE.L  D0,D1
00010002 4282                    CLR.L   D2
00010004 D401                    ADD.B   D1,D2
00010006 E289                    LSR.L   #1,D1
00010008 66FA                    BNE.B   $10004
0001000A E20A                    LSR.B   #1,D2
0001000C 55C2                    SCS     D2
0001000E 60FE                    BRA.B   $1000E
020Bug> RM D0 <CR>
```

Initialize D0 and start target program:

```
D0  =00000000 ?  52A9C._<CR>
020Bug> GD 10000 <CR>
Effective address: 00010000
```

To exit target code, press ABORT pushbutton.

```
Exception: Abort
Format Vector = 0100
PC  =0001000E SR  =2711=TR:OFF S. 7 X...C
USP =0000F830 MSP =0000FC18 ISP*=0000FFF8 VBR =00000000
SFC =0=XX     DFC =0=XX     CACR=0=..     CAAR=00000000
D0  =00052A9C D1  =00000000 D2  =000000FF D3  =00000000
D4  =00000000 D5  =00000000 D6  =00000000 D7  =00000000
A0  =00000000 A1  =00000000 A2  =00000000 A3  =00000000
A4  =00000000 A5  =00000000 A6  =00000000 A7  =0000FFF8
```

Set PC to start of program and restart target code:

```
020Bug> RM PC <CR>
PC  =0001000E ?  10000._<CR>
020Bug> GD <CR>
Effective address: 00010000
```

GN

GN sets a temporary breakpoint at the address of the next instruction, that is, the one following the current instruction, and then starts target code execution. After setting the temporary breakpoint, the sequence of events is similar to that of the GO command.

GN is especially helpful when debugging modular code because it allows the user to "trace" through a subroutine call as if it were a single instruction.

Example: The following section of code resides at address $2000.

```
020Bug> MD 2000:4;DI <CR>
00002000 7003                           MOVE.L   #3,D0
00002002 7201                           MOVEQ.L  #1,D1
00002004 6100FFA                        BSR.W    $3000
00002008 2600                           MOVE.L   D0,D3
020Bug>
```

The following simple subroutine resides at address $3000.

```
020Bug> MD 3000:2;DI <CR>
00003000 D081                           ADD.L  D1,D0
00003002 4E75                           RTS
020Bug>
```

Execute up to the BSR instruction.

```
020Bug> RM PC <CR>
PC  =00000000 ?  2000._<CR>
020Bug> GT 2004 <CR>
Effective address: 00002004
Effective address: 00002000
At Breakpoint
PC  =00002004 SR  =2700=TR:OFF S.7  .....
USP =00003830 MSP =00003C18 ISP*=00004000 VBR =00000000
SFC =0=XX     DFC =0=XX     CACR=0=..     CAAR=00000000
D0  =00000003 D1  =00000001 D2  =00000000 D3  =00000000
D4  =00000000 D5  =00000000 D6  =00000000 D7  =00000000
A0  =00000000 A1  =00000000 A2  =00000000 A3  =00000000
A4  =00000000 A5  =00000000 A6  =00000000 A7  =00004000
00002004 61000FFA                   BSR.W  $3000
020Bug>
```

Use the GN command to "trace" through the subroutine call and display the results.

```
020Bug> GN <CR>
Effective address: 00002008
Effective address: 00002004
At Breakpoint
```

```
PC  =00002008 SR  =2700=TR:OFF S.7  .....
USP =00003830 MSP =00003C18 ISP*=00004000 VBR =00000000
SFC =0=XX     DFC =0=XX     CACR=0=..     CAAR=00000000
D0  =00000004 D1  =00000001 D2  =00000000 D3  =00000000
D4  =00000000 D5  =00000000 D6  =00000000 D7  =00000000
A0  =00000000 A1  =00000000 A2  =00000000 A3  =00000000
A4  =00000000 A5  =00000000 A6  =00000000 A7  =00004000
00002008 2600                        MOVE.L  D0,D3
020Bug>
```

3.10 GO EXECUTE USER PROGRAM GO

GO [<ADDR>]

The GO command (alias "G") is used to initiate target code execution. All previously set breakpoints are enabled. If an address is specified, it is placed in the target PC. Execution starts at the target PC address.

The sequence of events is as follows:
1) First, if an address is specified, it is loaded in the target PC.
2) Then, if a breakpoint is set at the target PC address, the instruction at the target PC is traced (executed in trace mode).
3) Next, all breakpoints are inserted in the target code.
4) Finally, target code execution resumes at the target PC address.

At this point control may be returned to 020Bug by various conditions:

1) A breakpoint with 0 count was found.
2) The user pressed ABORT or RESET.
3) An unexpected exception occurred.
4) By execution of the .RETURN TRAP #15 function.

Example: (The following program resides at $10000)

```
020Bug> MD 10000;DI <CR>
00010000 2200                    MOVE.L  D0,D1
00010002 4282                    CLR.L   D2
00010004 D401                    ADD.B   D1,D2
00010006 E289                    LSR.L   #1,D1
00010008 66FA                    BNE.B   $10004
0001000A E20A                    LSR.B   #1,D2
0001000C 55C2                    SCS     D2
0001000E 60FE                    BRA.B   $1000E
020Bug> RM D0 <CR>
```

Initialize D0, set some breakpoints, and start target program:

```
D0  =00000000 ?  52A9C._<CR>

020Bug> BR 10000,1000E <CR>
BREAKPOINTS
00010000             0001000E
020Bug> GO 10000 <CR>
Effective address: 00010000
At Breakpoint
PC  =0001000E SR  =2011=TR:OFF S. 0 X...C
USP =0000F830 MSP =0000FC18 ISP*=00010000 VBR =00000000
SFC =0=XX     DFC =0=XX     CACR=0=..     CAAR=00000000
D0  =00052A9C D1  =00000000 D2  =000000FF D3  =00000000
D4  =00000000 D5  =00000000 D6  =00000000 D7  =00000000
A0  =00000000 A1  =00000000 A2  =00000000 A3  =00000000
A4  =00000C00 A5  =00000000 A6  =00000000 A7  =00010000
0001000E 60FE                        BRA.B   $1000E
```

Note that in this case breakpoints are inserted after tracing the first instruction, therefore the first breakpoint is not taken.

Continue target program execution.

```
020Bug> G <CR>
Effective address: 0001000E
At Breakpoint
PC  =0001000E SR  =2011=TR:OFF S. 0 X...C
USP =0000F830 MSP =0000FC18 ISP*=00010000 VBR =00000000
SFC =0=XX     DFC =0=XX     CACR=0=..     CAAR=00000000
D0  =00052A9C D1  =00000000 D2  =000000FF D3  =00000000
D4  =00000000 D5  =00000000 D6  =00000000 D7  =00000000
A0  =00000000 A1  =00000000 A2  =00000000 A3  =00000000
A4  =00000000 A5  =00000000 A6  =00000000 A7  =00010000
0001000E 60FE                       BRA.B  $1000E
```

Remove breakpoints and restart target code.

```
020Bug> NOBR <CR>
BREAKPOINTS 020Bug> GO 10000 <CR>
Effective address: 00010000
```

To exit target code, press the ABORT pushbutton.

```
Exception: Abort
Format Vector = 0100
PC  =0001000E SR  =2011=TR:OFF S. 0 X...C
USP =0000F830 MSP =0000FC18 ISP*=0000FFF8 VBR =00000000
SFC =0=XX     DFC =0=XX     CACR=0=..     CAAR=00000000
D0  =000529AC D1  =00000000 D2  =000000FF D3  =00000000
D4  =00000000 D5  =00000000 D6  =00000000 D7  =00000000
A0  =00000000 A1  =00000000 A2  =00000000 A3  =00000000
A4  =00000000 A5  =00000000 A6  =00000000 A7  =0000FFF8
```

3.11 GO TO TEMPORARY BREAKPOINT GT

GT <ADDR>

GT allows the user to set a temporary breakpoint and then start target code execution. A count may be specified with the temporary breakpoint. Control is given at the target PC address. All previously set breakpoints are enabled. The temporary breakpoint is removed when any breakpoint with 0 count is encountered.

After setting the temporary breakpoint, the sequence of events is similar to that of the GO command. At this point control may be returned to 020Bug by various conditions:

1) A breakpoint with 0 count was found.
2) The user pressed ABORT or RESET.
3) An unexpected exception occurred.
4) By execution of the .RETURN TRAP #15 function.

Example: (The following program resides at $2000)

```
020Bug> MD 2000;DI <CR>
00002000 2200                    MOVE.L  D0,D1
00002002 4282                    CLR.L   D2
00002004 D401                    ADD.B   D1,D2
00002006 E289                    LSR.L   #1,D1
00002008 66FA                    BNE.B   $2004
0000200A E20A                    LSR.B   #1,D2
0000200C 55C2                    SCS     D2
0000200E 60FE                    BRA.B   $200E
020Bug> RM D0 <CR>
```

Initialize D0 and set a breakpoint:

```
D0  =00000000 ?  52A9C._<CR>

020Bug> BR 200E <CR>
BREAKPOINTS
0001000E
020Bug>
```

Set PC to start of program, set temporary breakpoint, and start target code:

```
020Bug> RM PC <CR>
PC  =0001000E ?  2000._<CR>

020Bug> GT 2006 <CR>
Effective address: 0002006
Effective address: 0002000
At Breakpoint
PC  =00002006 SR  =2711=TR:OFF S. 7 X...C
USP =00003830 MSP =00003C18 ISP*=00004000 VBR =00000000
SFC =0=XX     DFC =0=XX     CACR=0=..     CAAR=00000000
```

```
D0  =00052A9C D1  =00000029 D2  =00000009 D3  =00000000
D4  =00000000 D5  =00000000 D6  =00000000 D7  =00000000
A0  =00000000 A1  =00000000 A2  =00000000 A3  =00000000
A4  =00000000 A5  =00000000 A6  =00000000 A7  =00004000
00002006 E289                        LSR.L  #1,D1
020Bug>
```

Set another temporary breakpoint at $10002 and continue the target program execution.

```
020Bug> GT 2002 <CR>
Effective address: 00010006
At Breakpoint
PC  =0000200E SR  =2711=TR:OFF S. 7 X...C
USP =00003830 MSP =00003C18 ISP*=00004000 VBR =00000000
SFC =0=XX     DFC =0=XX     CACR=0=..     CAAR=00000000
D0  =00052A9C D1  =00000000 D2  =000000FF D3  =00000000
D4  =00000000 D5  =00000000 D6  =00000000 D7  =00000000
A0  =00000000 A1  =00000000 A2  =00000000 A3  =00000000
A4  =00000000 A5  =00000000 A6  =00000000 A7  =00004000
0000200E 60FE                         BRA.B   $200E
020Bug>
```

Note that a breakpoint from the breakpoint table was encountered before the temporary breakpoint.

3.12 HELP HE

HE [<COMMAND>]

HE is the 020Bug help facility. HE <CR> displays the command name of all available commands along with a brief description of each one. HE <COMMAND> displays only the command name and description for that particular command.

Example: 020Bug> HE <CR>

```
MD      Memory Display
MM      Memory Modify
M       "Alias" for previous command
MS      Memory Set
RD      RegisterDisplay
RM      Register Modify
DC      Data Conversion and Expression Evaluation
BR      Breakpoint Insert
NOBR    Breakpoint Delete
T       Trace Instruction
TC      Trace on Change of Flow
TT      Trace to Temporary Breakpoint
GO      GO to Target Code
G       "Alias" for previous command
GT      Go and Insert Temporary Breakpoint
GD      Go Direct (no breakpoints)
GN      Go and Stop after Next Instruction
BF      Block Fill
BM      Block Move
BS      Block Search
LO      Load S-Records
VE      Verify S-Records
DU      Dump S-Records
TM      Transparent Mode
OF      Offset Registers
OS      Switch to Operating System
PA      Printer Attach
NOPA    Printer Detach
PF      Port Format
SD      Switch Directory
TS      Set time-of-day clock
TD      Display time-of-day clock
020Bug> HE TC <CR>
TC      Trace on Change of Flow
020Bug>
```

3.13 LOAD S-RECORDS FROM HOST LO

LO [<ADDR>] [;<X/-C>] [=<text>]

This command is used when data in the form of a file of Motorola S-Records is to be downloaded from a host system to the GMX Micro-20. The LO command accepts serial data from the host and loads it into memory.

The optional <ADDR> field allows the user to enter an offset address which is to be added to the address contained in the address field of each record. This will cause the records to be stored to memory at different locations then would normally occur. The contents of the automatic offset register are not added to the S-Record addresses.

The optional text field, entered after the equals sign (=), will be sent to the host before 020Bug begins to look for S-Records at the host port. This allows the user to send a command to the host device to initiate the download. This text should NOT be delimited by any kind of quote marks. The text is understood to begin immediately following the equals sign and terminate with the carriage return. If the host is operating full duplex, the string will also be echoed back to the host port by the host and will appear on the user's terminal screen.

In order to accommodate host systems that echo all received characters, the above-mentioned text string is sent to the host one character at a time and characters received from the host are read one at a time. After the entire command has been sent to the host LO will keep looking for a LF character from the host, signifying the end of the echoed command. No data records will be processed until this LF is received. If the host system does not echo characters, LO will still keep looking for a LF character before data records are processed. For this reason it is required in situations where the host system does not echo characters that the first record transferred by the host system be a header record. The header record is not used but the LF after the header record serves to break LO out of the loop so that data records will be processed.

The other options have the following effects:

-C option - Ignore checksum. A checksum for the data contained within an SRecord is read in at the port. Normally, this calculated checksum is compared to the checksum contained within the S-Record and if the compare fails an error message is sent to the screen on completion of the download. If this option is selected then the comparison is not made.

X option - Echo. Echoes the S-Records to the user's terminal as they are read in at the host port.

The S-Record format (see Appendix C) allows for an entry point to be specified in the address field of the termination record of an S-Record block. If the address field of the termination record contains an address other than zero then that address (plus the offset address, if any) will be put into the target PC. Thus after a download the user need only enter G or GO instead of G <addr> or GO <addr> to execute the code that was downloaded.

If a non-hex character is encountered within the data field of a data record then the part of the record which had been received up to that time will be

printed to the screen and 020Bug's error handler will be invoked to point to the faulty character.

As mentioned, if the embedded checksum of a record does not agree with the checksum calculated by 020Bug AND if the checksum comparison has not been disabled via the "-C" option then an error condition exists. A message will be output stating the address of the record (as obtained from the address field of the record), the calculated checksum and the checksum read with the record. A copy of the record is also output. This is a fatal error and causes the command to abort.

When a load is in progress, each data byte is written to memory and then the contents of this memory location are compared to the data to determine if the data stored properly. If for some reason the compare fails then a message is output stating the address where the data was to be stored, the data written and the data read back during the compare. This is also a fatal error and will cause the command to abort.

Since processing of the S-Records is done character-by-character, any data that was deemed good will have already been stored to memory if the command aborts due to an error.

Examples:

Suppose a host system (a VME/10 with VERSAdos in this case) was used to create a program that looks like this:

```
 1                          *  Test Program.
 2                          *
 3             65040000                ORG       $65040000
 4
 5    65040000 7001                     MOVEQ.L   #1,D0
 6    65040002 D088                     ADD.L     A0,D0
 7    65040004 4A00                     TST.B     D0
 8    65040006 4E75                     RTS
 9                                      END

******  TOTAL ERRORS      0--
******  TOTAL WARNINGS    0--
```

Then this program was converted into an S-Record file named TEST.MX that looks like this:

```
S00F000054455354533353372020010 15E
S30D65040000700 1D0884A004E75B3
S7056504000091
```

Load this file into the GMX Micro-20 memory for execution at address $40000 as follows:

```
020Bug>TM_<CR>                  ( Go into transparent mode to establish      )
Escape character: $01=  A       ( communication with the VME/10.             )
<BREAK>                         ( Press BREAK key to get VERSAdos login      )
                                ( prompt.                                    )
```

"

```
 (login)                          ( User must log onto VERSAdos and enter the   )
   "                              ( proper catalog to access the file TEST.MX   )
   "
= <^A>                           ( Enter escape character to return to         )
                                 (020Bug prompt.                               )
```

```
020Bug> LO -65000000 ;X=COPY TEST.MX,#_<CR>
COPY TEST.MX,#
S30D6504000070010D0884A004E75B3
S7056504000091
020Bug>
```

The S-Records are echoed to the terminal because of the "X" option. Note that the S0 header record is not echoed.

The offset address of -65000000 was added to the addresses of the records in FILE.MX and caused the program to be loaded to memory starting at $40000. The text "COPY TEST.MX,#" is a VERSAdos command line that caused the file to be copied by VERSAdos to the VME/10 port which is connected with the GMX Micro-20's host port.

```
020Bug> MD 40000:4;DI <CR>
00040000 7001                    MOVEQ.L  #1,D0
00040002 D088                    ADD.L    A0,D0
00040004 4A00                    TST.B    D0
00040006 4E75                    RTS
020Bug>
```

The target PC now contains the entry point of the code in memory ($40000).

3.14 MEMORY DISPLAY MD

MD[S] <ADDR>[:<COUNT> : <ADDR>][; [B:W:L:S:D:X:P:DI]]

This command is used to display the contents of multiple memory locations all at once. MD accepts the following data types:

B - Byte
W - Word
L - Longword
S - Single precision floating-point
D - Double precision floating-point
X - eXtended precision floating-point
P - Packed decimal floating-point

The default data size is word. Integer data is displayed in hex; floating-point data is displayed in hex and also in decimal scientific notation. The DI option selects the built-in 020Bug disassembler. No other option is allowed if DI is selected.

The optional count argument in the MD command specifies the number of data items to be displayed (or the number of instructions to disassemble if the disassembly option is selected). The default value is 8 if none is entered. Entering only <CR> at the prompt immediately after the command has completed will cause the command to re-execute, displaying an equal number of data items or lines beginning at the next address.

Example 1:

```
020Bug> md 12000 <CR>
00012000 2800  1942  2900  1942   2800  1842  2900  2846 (..B)..B(..B)..F
020Bug> <CR>
00012010 FC20  0050  ED07  9F61   FF00  000A  E860  F060 : .Pm..a....h'p'
```

Example 2: Assume the following processor state: A2=00013500,D5=53F00127

```
020Bug> md (a2,d5):&19;b <CR>
00013627 4F 82 00 C5 9B 10 33 7A DF 01 6C 3D 4B 50 0F 0F      O..E..3z_.l=KP..
00013637 31 AB 80                                             +1.
020Bug>
```

Example 3:

```
020Bug> md 50008;di <CR>
00050008 46FC2700              MOVE.W   #9984,SR
0005000C 61FF0000023E          BSR.L    $5024C
00050012 4E7AD801              MOVEC.L  VBR,A5
00050016 41ED7FFC              LEA.L    32764(A5),A0
0005001A 5888                  ADDQ.L   #4,A0
0005001C 2E48                  MOVE.L   A0,A7
0005001E 2C48                  MOVE.L   A0,A6
00050020 13C7FFFB003A          MOVE.B   D7,($FFFB003A).L
020Bug>
```

Example 4:

```
020Bug> md 6000;X <CR>
00060000 0_50C8_4E81D1514C3B4682= 2.05868705929099575E-293
0006000C 0_49F4_47AE30E44567A615= 1.1848124282926256_E+767
00060018 0_4321_40DA90773E9405A8= 1.3513932923024227_E+241
00060024 0_3C4D_3A06F00A37C0653B= 7.6215797862099370_E-286
00060030 1_3579_333349FD30ECC4CE=-4.2389212673767906_E-812
0006003C 0_2EA6_2C5FAF302A192461= 4.6313284761559737_E-338
00060048 0_67D2_258C0EC3234583F4= 2.9388148G82109845_E-068
00060054 0_20FE_1EB86E561C71E387= 1.2739189165036265_E-390
```

MM <ADDR> [;[[N][B|W|L|S|D|X|P|A]|DI]

This command is used to examine and change memory locations. MM accepts the following data types:

B - Byte
W - Word
L - Longword
S - Single precision floating-point
D - Double precision floating-point
X - eXtended precision floating-point
P - Packed decimal floating-point

The default data type is word. The MM command (alias "M") reads and displays the contents of memory at the specified address and prompts the user with a question mark ("?"). The user may enter new data for the memory location, followed by <CR>, or may simply enter <CR>, which leaves the contents unaltered. That memory location will be closed and the next memory location will be opened.

The user may also enter one of four special characters, either at the prompt or at the end of an input data string, which change what happens when carriage return is entered. These special characters are as follows:

"V" or "v" - The next successive memory location will be opened. (This is the default. It is in effect whenever MM is invoked and remains in effect until changed by entering one of the other special characters).

"^" - MM will back up and open the previous memory location.

"=" - MM will re-open the same memory location (this is useful for examining I/O registers or memory locations that are changing over time.

"." - Terminates MM command. Control will return to 020Bug.

The N option of the MM command disables the read portion of the command. The A option forces alternate location accesses only.

If any floating-point format is selected, data may be entered in any of the input formats described in section 3.22 (Register Modify), but will be stored in the selected format. Because of the complexity of these formats, using one of the special characters after the data may cause the input to be rejected. If the input is single precision. double precision, extended precision, or packed hexadecimal, and the number of mantissa digits entered is less than the maximum allowed, or the input is in scientific notation, a trailing special character will cause the input to be rejected. The special character will still take effect.

Example 1:

```
020Bug> mm 10000 <CR>                Access location 10000
00010000 1234?  <CR>
00010002 5678?  4321 <CR>            Modify memory
```

```
00010004 9ABC?  8765^ <CR>          Modify memory and back up
00010002 4321?  <CR>
00010000 1234?  abcd._<CR>          Modify memory and exit
```

Example 2:

```
020Bug> mm 10001;1a <CR>            Longword access to location 10001
00010001 CD432187?  <CR>            (Alternate location accesses)
00010009 00068010?  68010+10=_<CR>  Modify and reopen location
00010009 00068020?  <CR>
00010009 00068020?  ._<CR>          Exit MM
```

Example 3:

```
020Bug> mm 78008;P_<CR>                                  Packed decimal
00078008 1000_816_356780258255045007  =_<CR>             Access same address
00078008 1000_816_356780258255045007  0000 023 625<CR>   packed decimal
00078008 0000_023_625000000000000007  0 25 3E0D9F<CR>    single precision
00078008 0100_027_119931191718441137  0 123 45FF5<CR>    double precision
00078008 0100_221_563660019126207007  1 35B3 BF00349<CR> extended precision
00078008 1100_794_455781455568081647  4.7893E50<CR>      scientific notation
00078008 0000_050_478900000000000007  &14<CR>            integer
00078008 0000_001_140000000000000007  0 25 3E0000v<CR>    single precision;
                                                         trailing 0s allow
                                                         step-to-next code
0007800C\ 0100_027_119931191718441137?  ._<CR>           Exit MM
```

The DI option enables the one-line assembler/disassembler. All other options are invalid if DI is selected. The contents of the specified memory location will be disassembled and displayed and the user will be prompted with a question mark ("?") for input. At this point the user has three options:

1) Enter <CR> . This will close the present location and will continue with disassembly of next instruction.

2) Enter a new source instruction followed by <CR>. This invokes the assembler, which will assemble the instruction and generate a "listing file" of one instruction.

3) Enter .<CR> . This will close the present location and will exit the MM command.

If a new source line is entered (#2 above), the present line will be erased and replaced by the new source line entered. If a hardcopy terminal is being used, port 0 should be reconfigured for hardcopy operation with the PF command. In the hardcopy mode, a line feed will be done instead of erasing the line.

If an error is found during assembly, the symbol "^" will appear below the field suspected of the error, followed by an error message. The location being accessed will be redisplayed.

For additional information about the assembler, see Chapter 4.

Example 4: Assemble a new source line.

```
020Bug> mm 10000;di <CR>
00010000 46FC2400                       MOVE.W  #9216,SR ?  divs.w -(a2),d2 <CR>
00010000 85E2                           DIVS.W  -(A2),D2
00010002 2400                           MOVE.L   D0,D2?
```

Example 5: New source line with error.

```
00010008 4E7AD801                       MOVEC.L  VBR,A5 ?
bchg #$12,9(a5,d6)) <CR>
00010008                                BCHG     #$12,9(A5,D6))
--------------------------------------------------------------^
***Unknown Field ***

00010008 4E7AD801                       MOVEC.L   VBR,A5 ?
```

Example 6: Step to next location and exit MM.

```
020Bug> m 1000c;di <CR>
0001000C 000000FF                       OR.B    #255,D0 ?  <CR>
00010010 20C9                           MOVE.L  A1,(A0)+ ?  .<CR>
020Bug>
```

MS <ADDR> Hexadecimal number / 'string'

Memory Set is used to write data to memory starting at the specified address. Hex numbers are not assumed to be of a particular size, so they can contain any number of digits (as allowed by command line buffer size). If an odd number of digits are entered, the least significant nybble of the last byte accessed will be unchanged.

ASCII strings can be entered by enclosing them in single quotes ('). To include a quote as part of the string two consecutive quotes should be entered.

Example: Assume that memory is initially cleared:

```
020Bug> ms 25000 0123456789abcDEF 'This is ''020Bug'''_ 23456 <CR>
020Bug> md 25000:20;b <CR>
00025000 01 23 45 67 89 AB CD EF   54 68 69 73 20 69 73 20      .#Eg.+MoThis is
00025010 27 30 32 30 42 75 67 27   23 45 60 00 00 00 00 00      '020Bug'#E'.....
020Bug>
```

3.17 OFFSET REGISTERS DISPLAY/MODIFY OF

OF [Rn[;A]]

OF allows the user to access and change pseudo-registers called offset registers. These registers are used to simplify the debugging of relocatable and position independent modules (see discussion about offset registers in section 2.1.1.2.2).

There are 8 offset registers (R0-R7), but only R0-R6 can be changed. R7 is always 0 and it is used to override the effect of the automatic register (see below).

Command usage:

OF - To display all offset registers. An asterisk indicates which register is the automatic register.

OF Rn - To display/modify Rn. The user can scroll through the registers in a way similar to that used by the MM command.

OF Rn;A - To display/modify Rn and set it as the automatic register. The automatic register is one that is automatically added to each absolute address argument of every command except if an offset register is explicitly added. An asterisk indicates which register is the automatic register.

Offset register rules:

1) At power up/reset R7 is the automatic register.

2) At power up/reset all offset registers are set to zero.

3) R7 is always zero, and cannot be changed.

4) Any offset register can be set as the automatic register.

5) The automatic register is always added to every absolute address argument of every 020Bug command, except when an offset register is explicitly added to an argument (see Table 2-1).

6) There is always an automatic register. Note that a convenient way to disable the effect of the automatic register is by setting R7 as the automatic register. (Note: This is the default condition, see item 1 above).

Examples:

Display offset registers.

```
020Bug> OF <CR>
R0  = 00000000 R1  =00000000 R2  =00000000 R3  =00000000
R4  = 00000000 R5  =00000000 R6  =00000000 R7* =00000000
```

Modify some offset registers.

```
020Bug> OF R0 <CR>
R0  =00000000? 20000 <CR>
R1  =00000000? 20000 ^<CR>
R0  =00020000? ._<CR>
```

Look at location $20000.

```
020Bug> M 20000;DI <CR>
00000+R0 41F954455354                LEA     ($54455354).L,A0 ._<CR>
020Bug> M 0+R0;DI <CR>
00000+R0 41F954455354                LEA     ($54455354).L,A0 ._<CR>
020Bug>
```

Set R0 as the automatic register.

```
020Bug> OF R0;A <CR>
R0*=00020000? ._<CR>
```

To look at location $20000.

```
020Bug> M 0;DI <CR>
00000+R0 41F954455354                LEA.L   ($54455354).L,A0 ._<CR>
020Bug>
```

To look at location 0, override the automatic offset.

```
020Bug> M 0+R7;DI <CR>
00000000                             DC.W    $FFF8 ._ <CR>
020Bug>
```

OS

This command causes control to be transferred to a program in the other part of the GMX Micro-20 ROM, which is normally the bootstrap for an operating system. A longword address at location $000014 in the ROM points to the start of this other program. The OS command causes a starting execution address and initial stack pointer value to be fetched from the first two longwords at the address pointed to by the value at $000014 in the ROM. Execution then begins with at the starting address with the new SP value in A7. Refer to Appendix A, "Alternate ROM Programs", for further explanation.

This function is normally used to initiate the bootstrap loader for an operating system supplied with the GMX Micro-20. However, the program started by the OS command could be anything at all that can be stored in the ROM, including a user-supplied program. Also note that the program started by the OS command is the same program which is automatically started in place of 020Bug if switch S1-1 is set OFF at RESET or Power-up, as described in Section 6 of the Hardware Setup Manual.

Example:

```
020Bug> OS <CR>
<....operating system startup message....>
```

PA
NOPA

These two commands "attach" or "detach" a serial printer from the GMX Micro-20 port 2. When the printer is attached, everything that appears on the system console terminal is also echoed to port 2 and is printed out by the printer. PA is used to attach the printer. NOPA is used to detach the printer.

The NOPA command detaches the printer such that activity at port 0 is no longer echoed to port 2. NOPA does not change the configuration of port 2. If no printer is attached when NOPA is invoked then the user receives a message to that effect.

The port characteristics that will be used when port 2 is configured for the printer can be examined and/or changed using the Port Format (PF) command.

Examples:

```
020Bug> PA <CR>
020Bug>

        ( The printer is now attached. Echoing will be done to the printer.   )
        ( This is useful for keeping a record of a debugging session. The     )
        ( user may also obtain a listing of a program which is in memory      )
        ( by invoking the MD command with the disassembly option. The         )
        ( disassembled source lines will sent to the printer as they are      )
        ( displayed on the terminal screen.                                   )

020Bug> NOPA <CR>
020Bug>

        ( The printer is now detached.                                        )

020Bug> NOPA <CR>
No printer was attached.
020Bug>
```

3.20 PORT FORMAT PF

PF[n]

Port format allows the user to configure the on-board serial ports. There are three serial ports accessible to 020BUG, ports 0, 1, and 2. Port 0 is the system console port, port 1 is the data link to a host system, and port 2 is the printer port. The parameters that can be changed for each port and their default values are given in the table below.

TABLE 3-2. PF COMMAND DEFAULT VALUES

Parameter	Port 0 Console	Port 1 Host	Port 2 Printer	Description
Hardcopy	N	---	---	Hardcopy console device
XON/XOFF	Y	Y	Y	XON/XOFF protocol
XON char	^Q	^Q	^Q	XON character
XOFF char	^S	^S	^S	XOFF character
Char Nulls	0	0	0	Number of nulls after every char
<CR> Nulls	0	0	0	Number of nulls after <CR>
Baud Rate (*)	19200	19200	19200	Baud rate
RTS/CTS	Y	Y	Y	RTS/CTS handshake
Parity	N	N	N	Parity generated and checked
Even/Odd	E	E	E	Generate odd or even parity
Bits/char	8	8	8	Bits per character
Stop bits	1	1	1	Stop bits after each character

(* Default if no baud rate is specified in ROM parameter area)

The baud rates available include 75, 110, 135, 150, 300, 600, 1200, 2000, 2400, 4800, 1800, 9600, 19200, and 38400. Use PFn To access the configuration table for port n. The changes to a configuration table will take effect after the last item has been entered. Use the BREAK key to abort the PF command and cancel any changes made to a configuration table

Examples:

```
020Bug> PF0 <CR>                          Access console configuration.
Hardcopy=N (Y/N)?  <CR>
XON/XOFF=Y (Y/N)?  <CR>
XON char :$11=^Q?  _^W <CR>               Change XON
XOFF char:$13=^S?  $20 <CR>               And XOFF characters
Char Nulls=&0?  <CR>
Nulls=&0?  <CR>
Baud Rate =&9600?  <CR>
RTS/CTS  =Y (Y/N) <CR>
Parity  =N (Y/N) Y                        Enable parity <CR>
Parity (E/O) =EVEN?  <CR>
Bits per character (5,6,7,8) =&8?  <CR>
Stop bits (1,2) =&1?  <CR>
```

```
020Bug> PF3 <CR>                                  Access printer configuration
XON/XOFF=Y (Y/N)?  <CR>
XON char :$11=^Q?  <CR>
XOFF char:$13=^S?  <CR>
Char Nulls=&0?  <CR>
Nulls=&0?  <CR>
Baud Rate =&300?  9600 <CR>                       Change baud rate
RTS/CTS  =Y (Y/N) <CR>
Parity  =N (Y/N) <CR>
Bits per character (5,6,7,8) =&8?  <CR>
Stop bits (1,2) =&1?  <CR>
020Bug>
```

Note: the system uses set #2 of the baud rates generated internally by the MC68681 DUART. This set includes 19,200 baud but not 38,400 baud. This latter rate is made available by programming the DUART's internal timer/counter to divide the master 3.6864 Mhz time base down to a 16x clock for 38,400 operation. If the user's application makes use of the DUART's timer/counter for any other purpose, the 38,400 baud rate cannot be used.

RD [+FPC]

The RD command is used to display the target state, that is, the processor state associated with the target program (see GO command). The instruction pointed to by the target PC is disassembled and displayed also. The processor registers are:

Number of registers		
8	A - Address Registers	(A0-A7)
8	D - Data Registers	(D0-D7)
10	S - System Registers	(PC,SR,USP,MSP,ISP,VBR,SFC,DFC,CACR,CAAR)

Note that A7 represents the active stack pointer, which leaves 25 different registers.

Example:

```
020Bug> RD <CR>
PC  =00008000 SR  =2705=TR:OFF S. 7 ..Z.C
USP =0000F830 MSP =0000FC18 ISP*=00010000 VBR =00000000
SFC =7=CS     DFC =1=UD     CACR=1=.E     CAAR=00000000
D0  =00000000 D1  =00000000 D2  =00000000 D3  =00000000
D4  =00000120 D5  =00000000 D6  =00000000 D7  =00000000
A0  =00000000 A1  =00000000 A2  =00000000 A3  =00009000
A4  =00000000 A5  =00000000 A6  =00000000 A7  =00010000
00008000 4AF34000                    TAS.B   0(A3,D4.W)
020Bug>
```

An asterisk following a stack pointer name indicates that it is the active stack pointer. The status register includes a mnemonic portion to help in reading it:

Trace Bits

T1	T0	Mnemonic	Description
0	0	TR:OFF	Trace off
0	1	TR:CHG	Trace on change of flow
1	0	TR:ALL	Trace all states
1	1	TR:INV	Invalid mode

S, M Bits: The bit name appears (S,M) if the respective bit is set, otherwise a "." indicates that it is cleared.

Interrupt Mask: A number from 0 to 7 indicates the current processor priority level.

Condition Codes: The bit name appears (X,N,Z,V,C) if the respective bit is set, otherwise a "." indicates that it is cleared.

The source and destination function code registers (SFC, DFC) include a two character mnemonic:

Function Code	Mnemonic	Description
0	XX	Undefined
1	UD	User Data
2	UP	User Program
3	XX	Undefined
4	XX	Undefined
5	SD	Supervisor Data
6	SP	Supervisor Program
7	CS	CPU Space

The CACR register shows mnemonics for two bits: Enable and Freeze. The bit name (E, F) appears if the respective bit is set, otherwise a "." indicates that it is cleared.

The RD command also can display the target state of the MC68881 Floating Point Coprocessor, which is saved and loaded just like the main processor's state (if an FPC is installed). Including "+FPC" on the command line enables this feature; "-FPC disables it. The FPC registers are

8	FP - Floating Point data registers	(FP0-FP7)
3	S - System Registers	(FCR, FSR, FAR)

Example:

```
020Bug> RD +FPC<CR>
PC  =00008000 SR  =2705=TR:OFF S. 7 ..Z.C
USP =0000F830 MSP =0000FC18 ISP*=00010000 VBR =00000000
SFC =7=CS     DFC =1=UD     CACR=1=.E     CAAR=00000000
D0  =00000000 D1  =00000000 D2  =00000000 D3  =00000000
D4  =00000120 D5  =00000000 D6  =00000000 D7  =00000000
A0  =00000000 A1  =00000000 A2  =00000000 A3  =00009000
A4  =00000000 A5  =00000000 A6  =00000000 A7  =00010000
FCR =00000000 FSR =0F000000-(CC=NZI[NAN]) FAR =00000000
FP0 =0_0000_0000000000000000= 0.0000000000000000_E+000
FP1 =0_0000_0000000000000000= 0.0000000000000000_E+000
FP2 =0_3FFF_9DF3B645A1CAC083= 1.2340000000000000_E+000
FP3 =0_0000_0000000000000000= 0.0000000000000000_E+000
FP4 =0_0000_0000000000000000= 0.0000000000000000_E+000
FP5 =0_3D33_9A28000000000000= 3.4936087938152341_E-216
FP6 =0_0000_0000000000000000= 0.0000000000000000_E+000
FP7 =0_0000_0000000000000000= 0.0000000000000000_E+000
00008000 4AF34000                    TAS.B   0(A3,D4.W)
```

The floating-point data registers are displayed as extended precision operands. For a complete explanation of floating point data formats the user should consult the MC68881 User's Manual. The hex display is organized as follows. The leading single digit represents the sign bit. The next block of four digits represents the 15-bit signed exponent. The third block (of 16 digits) represents the 64-bit mantissa.

Note that the Floating point Status Register (FSR) display is shown in mnemonic form as well as hexdecimal. The four FSR bits are the second digit in the hex display and represent the N, Z, I, and NAN bits of the FSR.

3.22 REGISTER MODIFY RM

RM <REG>

RM allows the user to display and change the target registers. It works in essentially the same way as the MM command, and the same special characters are used to control the display/change session (see MM command).

Example 1:

```
020Bug> RM D4 <CR>
D5  =12345678? ABCDEF<CR>          Modify register and backup
D4  =00000000? 3000._<CR>          Modify register and exit
020Bug>
```

Example 2:

```
020Bug> rm sfc <CR>
SFC =7=CS    ? 1=<CR>              Modify register and reopen
SFC =1=UD    ? .<CR>_              Exit
020Bug>
```

The RM command also can be used to examine and change the target registers of the MC68881 Floating Point Coprocessor if one is installed. Values for the three system registers of the FPC are entered as 32-bit hex values. However, only the most significant 8 bits of the FSR value are used; the other 24 bits of the FSR value are ignored. Values for the FPC data registers (FP0-FP7) may be entered in six different formats, which are shown in the following table. The numbers in parenthesis indicate the number of sign, exponent, and mantissa bits which will be extracted from the input string. The image at the right indicates the largest digits acceptable in each field. The underscore character is the only acceptable field separator, and underscores must be positioned exactly as shown or the input will be rejected. If the input string is accepted, it is converted to the appropriate format and stored in memory, then converted to extended precision by loading it into an FPC data register, and finally stored in the memory image of the selected register in extended precision format.

Single precision	(1,8,23)	1_FF_7FFFFF
Double precision	(1,11,52)	1_7FF_FFFFFFFFFFFF
Extended precision	(1,15,64)	1_7FFF_FFFFFFFFFFFFFFFF
Packed decimal	(4,12,68)	1111_999_99999999999999999
Scientific notation	(4,12,68)	[-]9.999988877776666[E[-]999]
Decimal integer	(32)	&999999999

In the first four formats, the first field is the sign bit(s), the second field is the exponent, and the last field is the mantissa. The number of sign digit(s) and exponent digits entered must be the same as shown here.

In packed decimal format, the first field of four single bits represents the two sign bits and two bits used to indicate NANs and infinities. The first bit is the mantissa sign bit, the second is the exponent sign bit, and the last two are the special flag bits.

Scientific notation input is converted to a packed decimal result. The special flag bits are always set to 0. The exponent field at the end is optional; if it is omitted, the exponent value defaults to 0. The sign characters in front of the mantissa and exponent are also optional; if omitted, the value is positive. Leading 0s may be omitted from the exponent field.

In all floating-point formats, all digits of the mantissa field except the first are optional; trailing 0s may be omitted. In decimal integer format, all digits after the first are optional; leading 0s may be omitted.

Example:

```
020Bug> RM FSR <CR>
FSR =00000000-(CC=....    ) ?  F000000 <CR>
FAR =00000000 ?  _<CR>
FP0 =0_0000_0000000000000000= 0.0000000000000000_E+000?  1 22 333333 <CR>
FP1 =0_3D33_9A28000000000000= 3.4936087938152341_E-216?   0 2E4 987123DEFAB <CR>
FP2 =0_3FFF_9DF3B645A1CAC083= 1.2340000000000000_E+000?  1 0088 373F <CR>
FP3 =0_0000_0000000000000000= 0.0000000000000000_E+000?  0100 012 3930033 <CR>
FP4 =0_0000_0000000000000000= 0.0000000000000000_E+000?  1.234E-48 <CR>
FP5 =0_0000_0000000000000000= 0.0000000000000000_E+000?  6.98765 <CR>
FP6 =0_0000_0000000000000000= 0.0000000000000000_E+000?  &3499045 <CR>
FP7 =0_0000_0000000000000000= 0.0000000000000000_E+000?  ._<CR>

020Bug> RD <CR>
PC  =00008000 SR  =2705=TR:OFF S. 7 ..Z.C
USP =0000F830 MSP =0000FC18 ISP*=00010000 VBR =00000000
SFC =7=CS     DFC =1=UD     CACR=1=.E     CAAR=00000000
D0  =00000000 D1  =00000000 D2  =00000000 D3  =00000000
D4  =00000120 D5  =00000000 D6  =00000000 D7  =00000000
A0  =00000000 A1  =00000000 A2  =00000000 A3  =00009000
A4  =00000000 A5  =00000000 A6  =00000000 A7  =00010000
FCR =00000000 FSR =0F000000-(CC=NZI[NAN]) FAR =00000000
FP0 =1_3FA2_B333330000000000=-1.4136387180819289_E-028
FP1 =0_3EE4_CC3891EF7D580000= 1.0266008193855691_E+000
FP2 =1_0088_373F000000000000=-6.3205089947598130_E-892
FP3 =0_3FD9_8A46971E78B1462F= 3.9303300000000000_E-012
FP4 =0_3F5F_E6D8DBFABBCC0831= 1.2340000000000000_E-048
FP5 =0_4001_DF9AD42C3C9EECC0= 6.9876500000000000_E+000
FP6 =0_4014_D590940000000000= 3.4990450000000000_E+006
FP7 =0_0000_0000000000000000= 0.0000000000000000_E+000
00008000 4AF34000                    TAS.B   0(A3,D4.W)
```

RS

This command is causes the system to be restarted as if the RESET button had been pressed. The stack pointer is reset to the power-up value and execution starts at the values in the power-up vectors at the begining of the ROMs. The stack value is in $800000, and the execution address is in $800004. The confidence test is performed, and a RESET instruction is executed, which causes the CPU to send a RESET signal to the rest of the system.

Example:

```
020Bug> RS<CR>
GMX Micro-20 Debugger/Diagnostics Version 2.41 - 7/17/86
THURSDAY  01/23/87   12:21:42
020Bug>
```

If the system is configured for automatic self-test operation (see section 1.4), entering the RS command will cause self-test operation to be restarted.

SD

This command is used to change from the debugger directory to the diagnostic directory or from the diagnostic directory to the debugger directory.

The commands in the current directory (the directory that the user is in at the particular time) may be listed using the help (HE) command.

The way the directories are structured, the debugger commands are available from either directory but the diagnostic commands are only available from the diagnostic directory.

Example 1:

```
020Bug> SD <CR>
M20Diag>                    ( The user has changed from the debugger    )
                            ( directory to the diagnostic directory,    )
                            ( as can be seen by the "M20Diag>" prompt.  )
```

Example 2:

```
M20Diag> SD <CR>
020Bug>                     ( The user is now back in the debugger      )
                            ( directory.                                )
```

3.25 TRACE T

T [<HEX COUNT>]

The T command allows execution of one instruction at a time, displaying the target state after execution. T starts tracing at the address in the target PC. The optional count field (which defaults to 1 if none entered) specifies the number of instructions to be traced before returning control to 020Bug.

Breakpoints are monitored (but not inserted) during tracing for all trace commands, which allows the use of breakpoints in ROM or write protected memory. In all cases, if a breakpoint with 0 count is encountered, control will be returned to 020Bug.

The trace functions are implemented with the trace bits (T0, T1) in the MC68020 status register, therefore, these bits should not be modified by the user while using the trace commands.

Example: (The following program resides at location $10000)

```
020Bug> MD 10000;DI <CR>
00010000 2200                           MOVE.L  D0,D1
00010002 4282                           CLR.L   D2
00010004 D401                           ADD.B   D1,D2
00010006 E289                           LSR.L   #1,D1
00010008 66FA                           BNE.B   $10004
0001000A E20A                           LSR.B   #1,D2
0001000C 55C2                           SCS     D2
0001000E 60FE                           BRA.B   $1000E
020Bug>
```

Initialize PC and D0:

```
020Bug> RM PC <CR>
PC  =00008000 ?  10000.
020Bug> RM D0 <CR>
D0  =00000000 ?  8F41C.
```

Display target registers and trace one instruction:

```
020Bug> RD <CR>
PC  =00010000 SR  =2700=TR:OFF S. 7 .....
USP =0000382C MSP =00003C14 ISP*=00004000 VBR =00000000
SFC =0=XX     DFC =0=XX     CACR=0=..     CAAR=00000000
D0  =0008F41C D1  =00000000 D2  =00000000 D3  =00000000
D4  =00000000 D5  =00000000 D6  =00000000 D7  =00000000
A0  =00000000 A1  =00000000 A2  =00000000 A3  =00000000
A4  =00000000 A5  =00000000 A6  =00000000 A7  =00004000
00010000 2200                       MOVE.L  D0,D1
020Bug> T <CR>
```

```
PC  =00010002 SR  =2700=TR:OFF S. 7 .....
USP =0000382C MSP =00003C14 ISP*=00004000 VBR =00000000
SFC =0=XX     DFC =0=XX     CACR=0=..     CAAR=00000000
D0  =0008F41C D1  =0008F41C D2  =00000000 D3  =00000000
D4  =00000000 D5  =00000000 D6  =00000000 D7  =00000000
A0  =00000000 A1  =00000000 A2  =00000000 A3  =00000000
A4  =00000000 A5  =00000000 A6  =00000000 A7  =00004000
00010002 4282                        CLR.L   D2
020Bug>
```

Trace next instruction:

```
020Bug> <CR>
PC  =00010004 SR  =2700=TR:OFF S. 7 ..Z..
USP =0000382C MSP =00003C14 ISP*=00004000 VBR =00000000
SFC =0=XX     DFC =0=XX     CACR=0=..     CAAR=00000000
D0  =0008F41C D1  =0008F41C D2  =00000000 D3  =00000000
D4  =00000000 D5  =00000000 D6  =00000000 D7  =00000000
A0  =00000000 A1  =00000000 A2  =00000000 A3  =00000000
A4  =00000000 A5  =00000000 A6  =00000000 A7  =00004000
00010004 D401                         ADD.B   D1,D2
020Bug>
```

Trace the next two instructions:

```
020Bug> T 2 <CR>
PC  =00010006 SR  =2700=TR:OFF S. 7 .....
USP =0000382C MSP =00003C14 ISP*=00004000 VBR =00000000
SFC =0=XX     DFC =0=XX     CACR=0=..     CAAR=00000000
D0  =0008F41C D1  =0008F41C D2  =0000001C D3  =00000000
D4  =00000000 D5  =00000000 D6  =00000000 D7  =00000000
A0  =00000000 A1  =00000000 A2  =00000000 A3  =00000000
A4  =00000000 A5  =00000000 A6  =00000000 A7  =00004000
00010006 E289                         LSR.L  #1,D1
PC  =00010008 SR  =2700=TR:OFF S. 7 .....
USP =0000382C MSP =00003C14 ISP*=00004000 VBR =00000000
SFC =0=XX     DFC =0=XX     CACR=0=..     CAAR=00000000
D0  =0008F41C D1  =00047A0E D2  =0000001C D3  =00000000
D4  =00000000 D5  =00000000 D6  =00000000 D7  =00000000
A0  =00000000 A1  =00000000 A2  =00000000 A3  =00000000
A4  =00000000 A5  =00000000 A6  =00000000 A7  =00004000
00010008 66FA                         BNE.B   $10004
020Bug>
```

3.26 TRACE ON CHANGE OF CONTROL FLOW TC

TC [<COUNT>]

TC will start execution at the address in the target PC and will begin tracing upon the detection of an instruction that causes a change of control flow, such as JSR, BSR, RTS, etc. This means that execution will be in real time until a change of flow instruction is encountered. The optional count field (which defaults to 1 if none entered) specifies the number of change of flow instructions to be traced before returning control to 020Bug.

Breakpoints are monitored (but not inserted) during tracing for all trace commands, which allows the use of breakpoints in ROM or write protected memory. Note that the TC command will recognize a breakpoint only if it is at a change of flow instruction. In all cases, if a breakpoint with 0 count is encountered, control will be returned to 020Bug.

The trace functions are implemented with the trace bits (T0, T1) in the MC68020 status register, therefore, these bits should not be modified by the user while using the trace commands.

Example: (The following program resides at location $10000)

```
020Bug> MD 10000;DI <CR>
00010000 2200                           MOVE.L  D0,D1
00010002 4282                           CLR.L   D2
00010004 D401                           ADD.B   D1,D2
00010006 E289                           LSR.L   #1,D1
00010008 66FA                           BNE.B   $10004
0001000A E20A                           LSR.B   #1,D2
0001000C 55C2                           SCS     D2
0001000E 60FE                           BRA.B   $1000E
020Bug>
```

Initialize PC and D0:

```
020Bug> RM PC <CR>
PC  =00008000 ?  10000.
020Bug> RM D0 <CR>
D0 =00000000 ?  8F41C.
```

Trace on change of flow:

```
020Bug> TC <CR>
00010008 66FA                           BNE.B   $10004
PC  =00010004 SR  =2700=TR:OFF S. 7 .....
USP =0000382C MSP =00003C14 ISP*=00004000 VBR =00000000
SFC =0=XX     DFC =0=XX     CACR=0=..     CAAR=00000000
D0  =0008F41C D1  =00047A0E D2  =0000001C D3  =00000000
D4  =00000000 D5  =00000000 D6  =00000000 D7  =00000000
A0  =00000000 A1  =00000000 A2  =00000000 A3  =00000000
A4  =00000000 A5  =00000000 A6  =00000000 A7  =00004000
00010004 D401                       ADD.B   D1,D2
020Bug>
```

Note that the above display also shows the change of flow instruction.

TD

This command displays the current value in the GMX Micro-20's time-of-day clock/calendar.

Example:

```
020Bug> TD <CR>
THURSDAY  01/23/86   12:21:42
020Bug>
```

3.28 TRANSPARENT MODE TM

TM [<ESCAPE>]

TM essentially connects the two on-board serial ports (port 0 and port 1) together, allowing the user to communicate with a host computer. A message displayed by TM shows the current escape character, i.e., the character used to exit the transparent mode. The two ports remain "connected" until the escape character is received by port 0. The escape character is not transmitted to the host and at power up or reset is initialized to $01=^A.

The ports do not have to be at the same baud rate, but the terminal port baud rate should be equal to or greater than the host port baud rate for reliable operation. To change the baud rates use the PF command.

The optional escape argument allows the user to specify the character to be used as the exit character. This can be entered in three different formats:

```
ASCII code       :  $03       Set escape character to ^C
ASCII character  :  'c        Set escape character to "c"
control character:  ^c        Set escape character to ^C
```

Example 1:

```
020Bug> TM <CR>               Enter TM
Escape character:  $01=^A     Exit code is always displayed

<_^A>                         Exit transparent mode
```

Example 2:

```
020Bug> TM   ^g <CR>          Enter TM and set escape character
Escape character:  $07= ^G      to ^G

<_^G>                         Exit transparent mode
020Bug>
```

TS

TS allows the user to set the GMX Micro-20's hardware time-of-day clock and calendar to a new value. The current clock contents are displayed, and the user is prompted as to whether he wants to enter a new value. Each element of the time and date is displayed and prompted for separately. An empty line in response to a prompt defaults to the current value.

Example:

```
020Bug> TS <CR>

Current time value = THURSDAY  01/23/86   12:21:42

Set the clock (Y-N)?  N <CR>
020Bug> TS <CR>

Current time value = THURSDAY  01/23/86   12:21:49

Set the clock (Y-N)?  Y <CR>
Year    (00-99) = 86?  <CR>
Month   (01-12) = 01?  <CR>
Day     (01-31) = 23?  <CR>
Weekday (01-07, 1=Monday) = 4?  <CR>
Hour    (00-23) = 12?  <CR>
Minutes (00-59) = 21?  <CR>
Seconds (00-59) = 49?  55 <CR>
Current time value = THURSDAY  01/23/86   12:21:55

Is this time correct (Y-N)?  Y <CR>
```

3.30 TRACE TO TEMPORARY BREAKPOINT TT

TT <ADDR>

TT will set a temporary breakpoint at the specified address and will trace until a breakpoint with 0 count is encountered. The temporary breakpoint is then removed (TT is analogous to the GT command) and control is returned to 020Bug. Tracing starts at the target PC address.

Breakpoints are monitored (but not inserted) during tracing for all trace commands, which allows the use of breakpoints in ROM or write protected memory. If a breakpoint with 0 count is encountered, control will be returned to 020Bug.

The trace functions are implemented with the trace bits (T0, T1) in the MC68020 status register, therefore, these bits should not be modified by the user while using the trace commands.

Example: (The following program resides at location $10000)

```
020Bug> MD 10000;DI <CR>
00010000 2200                    MOVE.L  D0,D1
00010002 4282                    CLR.L   D2
00010004 D401                    ADD.B   D1,D2
00010006 E289                    LSR.L   #1,D1
00010008 66FA                    BNE.B   $10004
0001000A E20A                    LSR.B   #1,D2
0001000C 55C2                    SCS     D2
0001000E 60FE                    BRA.B   $1000E
020Bug>
```

Initialize PC and D0:

```
020Bug> RM PC <CR>
PC  =00008000 ?  10000.
020Bug> RM D0 <CR>
D0 =00000000 ?  8F41C.
```

Trace to temporary breakpoint:

```
020Bug> TT 10006 <CR>
PC  =00010002 SR  =2700=TR:OFF S. 7 .....
USP =0000382C MSP =00003C14 ISP*=00004000 VBR =00000000
SFC =0=XX     DFC =0=XX     CACR=0=..     CAAR=00000000
D0  =0008F41C D1  =0008F41C D2  =00000000 D3  =00000000
D4  =00000000 D5  =00000000 D6  =00000000 D7  =00000000
A0  =00000000 A1  =00000000 A2  =00000000 A3  =00000000
A4  =00000000 A5  =00000000 A6  =00000000 A7  =00004000
00010002 4282                        CLR.L  D2
PC  =00010004 SR  =2704=TR:OFF S. 7 ..Z..
USP =0000382C MSP =00003C14 ISP*=00004000 VBR =00000000
SFC =0=XX     DFC =0=XX     CACR=0=..     CAAR=00000000
D0  =0008F41C D1  =0008F41C D2  =00000000 D3  =00000000
D4  =00000000 D5  =00000000 D6  =00000000 D7  =00000000
A0  =00000000 A1  =00000000 A2  =00000000 A3  =00000000
A4  =00000000 A5  =00000000 A6  =00000000 A7  =00004000
00010004 D401                        ADD.B   D1,D2
```

```
At Breakpoint
PC  =00010006 SR  =2700=TR:OFF S. 7 .....
USP =0000382C MSP =00003C14 ISP*=00004000 VBR =00000000
SFC =0=XX     DFC =0=XX     CACR=0=..     CAAR=00000000
D0  =0008F41C D1  =0008F41C D2  =0000001C D3  =00000000
D4  =00000000 D5  =00000000 D6  =00000000 D7  =00000000
A0  =00000000 A1  =00000000 A2  =00000000 A3  =00000000
A4  =00000000 A5  =00000000 A6  =00000000 A7  =00004000
00010006 E289                        LSR.L   #1,D1
020Bug>
```

3.31 VERIFY S-RECORDS AGAINST MEMORY VE

VE [<ADDR>] [;<X/-C>] [=<text>]

This command is identical to the LO command with the exception that data is not stored to memory but merely compared to the contents of memory.

The VE command accepts serial data from a host system in the form of a file of Motorola S-Records and compares it to data already in the GMX Micro-20 memory. If the data does not compare then the user is alerted via information sent to the terminal screen.

The optional <ADDR> field allows the user to enter an offset address which is to be added to the address contained in the address field of each record. This will cause the records to be compared to memory at different locations then would normally occur. The contents of the automatic offset register are not added to the S-Record addresses.

The optional text field, entered after the equals sign (=), will be sent to the host before 020Bug begins to look for S-Records at the host port. This allows the user to send a command to the host device to initiate the download. This text should NOT be delimited by any kind of quote marks. The text is understood to begin immediately following the equals sign and terminate with the carriage return. If the host is operating full duplex, the string will also be echoed back to the host port by the host and will appear on the user's terminal screen.

In order to accommodate host systems that echo all received characters, the above-mentioned text string is sent to the host one character at a time and characters received from the host are read one at a time. After the entire command has been sent to the host LO will keep looking for a LF character from the host, signifying the end of the echoed command. No data records will be processed until this LF is received. If the host system does not echo characters, LO will still keep looking for a LF character before data records are processed. For this reason it is required in situations where the host system does not echo characters that the first record transferred by the host system be a header record. The header record is not used but the LF after the header record serves to break LO out of the loop so that data records will be processed.

The other options have the following effects:

-C option - Ignore checksum. A checksum for the data contained within an S-Record is calculated as the S-Record is read in at the port. Normally, this calculated checksum is compared to the checksum contained within the S-Record and if the compare fails an error message is sent to the screen on completion of the download. If this option is selected then the comparison is not made.

X option - Echo. Echoes the S-Records to the user's terminal as they are read in at the host port.

During a verify operation, an S-Record's data is compared to memory beginning with the address contained in the S-Record's address field (plus the offset address, if it was specified). If the verification fails then the non-comparing record is set aside until the verify is complete and then it is printed out to the screen. If three non-comparing records are encountered in the course of a

verify operation then the command is aborted.

If a non-hex character is encountered within the data field of a data record then the part of the record which had been received up to that time will be printed to the screen and 020Bug's error handler will be invoked to point to the faulty character.

As mentioned, if the embedded checksum of a record does not agree with the checksum calculated by 020Bug AND if the checksum comparison has not been disabled via the "-C" option then an error condition exists. A message will be output stating the address of the record (as obtained from the address field of the record), the calculated checksum and the checksum read with the record. A copy of the record is also output. This is a fatal error and causes the command to abort.

Examples:

This short program was developed on a host system.

```
1                           * Test Program.
2                           *
3              65040000             ORG      $65040000
4
5    65040000  7001                 MOVEQ.L  #1,D0
6    65040002  D088                 ADD.L    A0,D0
7    65040004  4A00                 TST.B    D0
8    65040006  4E75                 RTS
9                                   END

****** TOTAL ERRORS      0--
****** TOTAL WARNINGS    0--
```

Then this program was converted into an S-Record file named TEST.MX that looks like this:

```
S00F000054455354533353372020010 15E
S30D650400007001D0884A004E75B3
S7056504000091
```

This file was downloaded into memory at address $40000. The program may be examined in memory using the Memory Display command.

```
020Bug> MD 40000:4;DI <CR>
00040000 7001                       MOVEQ.L  #1,D0
00040002 D088                       ADD.L    A0,D0
00040004 4A00                       TST.B    D0
00040006 4E75                       RTS
020Bug>
```

Suppose that the user wants to make sure that the program has not been destroyed in memory. The VE command will be used to perform a verification.

```
020Bug> VE -65000000;X=COPY TEST.MX,#_<CR>
S30D650400007001D0884A004E75B3
S7056504000091
```

```
Verify passes.
020Bug>
```

The verification passes. The program stored in memory was the same as that in the S-Record file that had been downloaded. Now change the program in memory and perform the verification again.

```
020Bug> M 40002 <CR>
00040002 D088? D089. <CR>
020Bug> VE -65000000;X=COPY TEST.MX,# <CR>
S30D6504000070010D0884A004E75B3
S7056504000091

The following record(s) did not verify .....
S30D65040000------88--------B3
020Bug>
```

The byte which was changed in memory does not compare with the corrresponding byte in the S-Record.

CHAPTER 4

USING THE ONE-LINE ASSEMBLER/DISASSEMBLER

4.1 INTRODUCTION

The 020Bug firmware includes a one-line assembler and disassembler. The assembler is a mode of the Memory Modify command which accepts assembler source code lines and generates 68020 object code in the target memory. The disassembler is used by several commmands to display memory in the form of 68020 instruction mnemonics and associated operand information. Both the assembler and disassembler function on a line-by-line basis only.

The assembler supports all 68020 instructions except ILLEGAL, and also the DC.W directive for creating constants. The disassembler recognizes all 68020 instructions, and also all the instructions of the MC68881 Floating-Point Coprocessor (FPC), provided that the coprocessor ID number in the instruction is 1 (the Motorola default for the FPC). Other coprocessor instructions are disassemble in the generalized coprocessor instruction format described in the MC68020 User's Manual.

4.1.1 MC68020 Assembly Language

The symbolic language used to code programs for processing by the assembler is MC68020 assembly language. Assembly language consists of machine-instruction mnemonics, assembler directives (pseudo-ops), and operand expressions.

4.1.1.1 Machine-Instruction Mnemonics

The mnemonics for the MC68020 machine instructions are described in the MC68020 User's Manual (MC68020UM). The user should refer to this manual for the definition and description of each mnemonic.

4.1.1.2 Directives

Assembly language can contain mnemonic directives which specify actions to be performed by the assembler other than the generation of object code. The 020Bug assembler recognizes only the Define Constant Word directive, abbreviated "DC.W". This directive is used to define data within the program. Its use is explained in section 4.2.3.

4.1.1.3 Operand expressions

Nearly all MC68020 machine instructions operate on a specific object or objects. These operands may be defined as a register label, a constant value, a relative address, or a complex indirect addressing expression.

4.1.2 Comparison with MC68020 Resident Structured Assembler

There are several major differences between the 020Bug assembler and the MC68020 Resident Structured Assembler. The Resident assembler is a two-pass assembler

that processes an entire program as a unit, while the 020Bug assembler processes each line of a program independently. Because of this basic functional difference, the capabilities of the 020Bug assembler are more restricted:

1. Labels are not used or recognized. In the Resident assembler, label values are defined in pass 1, and used in pass 2 to reference lines and locations in a program. The one-line assembler has no information about any line of code other than the one being processed, and therefore has no way to associate a label with an address elsewhere in the program.

2. Source lines are not saved. In order to read back a program after it has been entered, the user must use the ;DI option of the MM and MD commands.

3. Only one directive is accepted (DC.W).

4. Macros cannot be defined or invoked.

5. No conditional assembly is used.

6. The "!", ">" and "<" symbols are recognized by the Resident assembler, but not by the 020Bug assembler. A leading ampersand character (&) specifies a decimal number when used with the 020Bug assembler (although numbers with no prefix are assumed to be decimal), but cannot represent a logical AND function as in the Resident assembler.

7. The Resident assembler can process arithmetic expressions including multiply and divide operations; the 020Bug assembler can only process add and subtract. The "/" character is only recognized as an item separator in register lists, and cannot be used as an arithmetic divide operator. The "*" character is recognized in arithmetic expressions as the location pointer; it is also used when a scale factor is to be included in an indexed address expression, i.e., "D0*4".

8. The Resident assembler accepts TDIV and TMUL as mnemonics for 32-bit multiply and divide with 32-bit results. The 020Bug assembler accepts the formats described in the MC68020 User's Manual for these instructions. This applies to both the signed and unsigned variants of these instructions.

Although functional differences exist between the two assemblers, the one-line assembler is a true subset of the Resident assembler. The format and syntax used with the 020Bug assembler are acceptable to the Resident assembler except as described in entries 6, 7, and 8 above.

4.2 SOURCE PROGRAM CODING

A source program is a sequence of source statements, which the assembler converts to object code, which can be executed by the processor to perform a predetermined task. Each source statement occupies a line and must be either an executable instruction or a DC.W assembler directive. Source statements must have the correct source line format.

4.2.1 Source Line Format

Each source statement is a combination of operation and, as required, operand fields. Line numbers, labels and comments are NOT used.

4.2.1.1 Operation Field

Since there is no label field, the operation field may begin in the first available column. It may also follow one or more spaces. The operation field may contain an instruction mnemonic corresponding to a MC68020 machine instruction, or a DC.W directive.

The size of the data object processed by an instruction is determined by the data size code. Some instructions can operate on more than one data size. For these operations, the data size code must be specified or a default size applicable to that instruction will be assumed. The size code need not be specified if only one data size is permitted by the operation. The data size code is specified by a period (.), appended to the operation field, and followed by B, W, or L, where:

B = byte integer	(8-bit data or displacement)
W = word integer	(16-bit data or displacement)
L = long word integer	(32-bit data or displacement)

No data size code is permitted when the instruction does not have a data size attribute. If no data size code is included with a sized instruction, the 020Bug assembler will default to word size.

Unlike the Resident assembler, the 020Bug assembler does not recognize the .S suffix as indicating an 8-bit displacement in branch instructions. The .B suffix must be used for this.

Examples (legal):

LEA	2(A0),A1	Longword size is assumed (.B, .W not allowed); this instruction loads the effective address of the first operand into A1.
ADD.B	(A0),D0	This instruction adds the byte whose address is (A0) to the lowest order byte in D0.
ADD	D1,D2	This instruction adds the low order word of D1 to the low order of D2. (W is the default size code.)
ADD.L	A3,D3	This instruction adds the entire 32-bit (longword) contents of A3 to D3.

Example (illegal):

SUBA.B	#5,A1	Illegal size specification (.B not allowed on SUBA). This instruction would have substracted the value 5 from the low order byte of A1; byte operations on address registers are not allowed.

4.2.1.2 Operand Field

If present, the operand field follows the operation field and is separated from the operation field by at least one space. When two or more operand subfields appear within a statement, they must be separated by a comma. In an instruction like 'ADD D1,D2' the first subfield (D1) is called the source effective address field (source <ea>), and the second subfield (D2) is called the destination effective address field (destination <ea>. o in this example, the contents of D1 are added to the contents of D2 and result is saved in D2. In the instruction 'MOVE D1,D2' the first subfield (D1) is the sending field and the second subfield (D2) is the receiving field. In other words, for most two-operand instructions, the general format 'operation source,destination' applies.

4.2.1.3 Mnemonics and Delimiters

The assembler recognizes all MC68020 instruction mnemonics except ILLEGAL. Numbers are recognized as both decimal and hexadecimal, with decimal the default case (note that this is reverse to the 020Bug commands):

a. Decimal - is a string of decimal digits (0-9) without a prefix (default) or preceded by an optional ampersand (&). Examples:

```
12334
&1234
```

b. Hexadecimal - is a string of hexadecimal digits (0-9, A-F) preceded by a dollar sign ($). Example:

```
$AFE5
```

One or more ASCII characters enclosed by apostrophes (') constitute an ASCII string. ASCII strings are left-justified and zero filled (if necessary), whether stored or used as immediate operands. This left justification will be to a word boundary if one or two characters are specified, or to a longword boundary if the string contains more than two characters.

```
005000      5300              DC.W     'S'
005002      223C41424344      MOVE.L   #'ABCD',D1
005008      3536              DC.W     '56'
```

The following register mnemonics are recognized by the assembler:

PC	Program Counter (only in PC-relative addressing)
SR	Status Register
CCR	Condition Codes Register (lower 8 bits of SR)
USP	User Stack Pointer
MSP	Master Stack Pointer

ISP Interrupt Stack Pointer

VBR Vector Base Register

SFC Source Function Code register

DFC Destination Function Code register

CACR CAche Control Register

CAAR CAche Address Register

D0-D7 Data registers 0-7

A0-A7 Address registers 0-7

Address register A7 represents the active system stack pointer, that is, one of USP, MSP, or ISP, as specified by the M and S bits of the status register (SR).

4.2.1.4 Character Set

The character set recognized by the 020Bug assembler is a subset of ASCII, as listed below:

1. The letters A through Z and a through z

2. The digits 0 through 9

3. Arithmetic operators: + -

4. Asterisk * (location pointer value, or scale factor code)

4. Parentheses () and brackets [] (used in address expressions)

5. Braces { (used in bitfield specifications)

5. Characters used as special prefixes:

    # (pound sign) specifies the immediate form of addressing.
    $ (dollar sign) specifies a hexadecimal number.
    & (ampersand) specifies a decimal number.
    @ (commercial at sign) specifies an octal number.
    % (percent sign) specifies a binary number.
    ' (apostrophe) specifies an ASCII literal character.

6. Six separating characters:

    Space
    , (comma)
    . (period)
    / (slash)
    - (dash)
    : (colon)

4.2.2 Addressing Modes

Effective addressing and data organization are described in detail in Section 2, "Data Organization and Addressing Capabilities", of the MC68020 User's Manual. Table 4-1 lists the addressing modes of the MC68020 which are accepted by the 020Bug one-line assembler. These expressions can be combined with an instruction mnemonic as a valid line of assembly source code.

TABLE 4-1. 020Bug ASSEMBLER ADDRESSING MODES

Format	Description
Dn	Data register direct
An	Address register direct
(An)	Address register indirect
(An)+	Address register indirect with postincrement
-(An)	Address register indirect with pre-increment
d(An) or (d,An)	Address register indirect with displacement
d(An,Rn) or (d,An,Rn)	Address register indirect with index
([bd],od)	Memory indirect (without base register, without index register)
([bd,An],od)	Memory indirect with base register, without index register)
([bd,An],Rn,od)	Memory indirect before indexing
([bd,An,Rn],od)	Memory indirect after indexing
(xxx).W	Absolute short
(xxx).L	Absolute long
LABEL(PC)	Program counter relative
LABEL(PC,Rn)	Program counter relative with index
([LABEL,ZPC])	PC relative memory indirect (PC suppressed, without index register)
([LABEL,PC])	PC relative memory indirect (without index register)
([LABEL,PC],Rn,od)	PC relative memory post-indexed
([LABEL,PC,Rn],od)	PC relative memory indirect pre-indexed
#xxx	Immediate

Note: Remember that this assembler/disassembler has no notion of labels. The "LABEL" field in the PC-relative addressing modes represents an expression.

4.2.3 DC.W Define Constant Word directive

The format for the DC.W directive is:

```
DC.W    <operand>
```

The function of the directive is to define a constant in memory. The DC.W directive can have only one operand (16-bit value) which can be the desired decimal, hexadecimal, or ASCII value, or an expression to be evaluated as a numeric value by the assembler. The constant is aligned on a word boundary, as word (.W) size is specified.

An ASCII string is recognized when characters are enclosed inside single quotes('). Each character (7 bits) is assigned to a byte of memory, with the eighth bit (MSB) always zero. If only one byte is entered, the data is left

justified in the word. A maximum of two ASCII characters may be entered with a DC.W directive.

Examples:

```
00010022    04D2    DC.W    1234      Decimal number
00010024    AAFE    DC.W    $AAFE     Hexadecimal number
00010026    4142    DC.W    'AB'      ASCII string
00010028    5443    DC.W    $5542+1   Expression
0001002A    4300    DC.W    'C'       ASCII character is left justified
```

4.3 DISASSEMBLING OBJECT CODE

020Bug has a facility for displaying the numeric object code of a program in memory as assembler source statements. This facility is used the Trace and Register Display commands, to show what instruction will be executed next, by the Memory Display command, when the DI option is selected, and by the Memory Modify command, when the DI option is selected.

The disassembler receives an address as a parameter, and assumes that the word at that address is an instruction word in a program. It will identify the instruction, and decode the effective address field(s) and extension word(s) if any. If the instruction is not a legal one, or the address modes or extension words are illegal in some way, the word will be decoded as a DC.W directive.

The output of the 020Bug disassembler looks like one line of a listing generated by the Resident assembler, and includes the base address of the instruction, its hexadecimal image, the instruction mnemonic and size code, and the operand value or address expression. Example:

```
ADDRESS  IMAGE                    MNEMONIC  OPERAND

00008A00 5468007B                 ADDQ.W    2,123(A0)
```

Numeric values generated by disassembly may be displayed in either hexadecimal or decimal, depending on the context. Address values, immediate values used for AND, OR, or EOR operations, immediate values moved to processor control registers, and immediate values used in floating point instructions are displayed in hexadecimal; all other values are displayed in decimal. Examples:

```
00008B00 028045612301             ANDI.L    $45612301,D0
00008B06 068045612301             ADDI.L    $116399357,D0
00008B0C 3028007B                 MOVE.W    123(A0),D0
00008B10 44FC007B                 MOVE.W    $7B,CCR
00008B14 303C007B                 MOVE.W    123,D0
00008B18 F23C441C9999AAAA         FACOS.S   $9999AAAA,FP0
```

If the instruction is a branch, or contains a PC-relative data reference, the displacement value contained in the instruction will be added to the base address of the instruction word or extension word to get the actual operand address, which is displayed in the operand field. Examples:

```
00008A04 67FA                     BEQ.B     $8A00

00008A1A 4A7A15E4                 TST.W     $0000A000(PC)
```

Up to ten bytes of hex image data will be displayed in the image field of the disassembly output line. If the instruction and its extension words exceeds ten bytes, a "+" will be displayed after the tenth byte, indicating more hex data not displayed. Example:

```
00007918 0CB00006DDD00170016+        CMPI.L     4500000,([($00160000).L,A0,ZD0.W*])
```

If the user has set up one or more of the 020Bug offset registers, any address which is in the range of one of these registers will be displayed in the form hhhhh+Rn, where Rn is the offset register the address is in the range of, and hhhhh is the displacement from the value of Rn to the address. This conversion is done for the instruction address at the beginning of the displacement line, and for all branch destination addresses. Other PC-relative addresses are displayed normally. Example:

```
020Bug>MD 7400:2;DI<cr>
00007400 603E                        BRA.B      $7440
00007402 4A7A8C2E                    TST.W      $00032(PC)
020Bug>OF R0<cr>
R0 =00000000? 7400<cr>
R1 =00000000?.<cr>
020Bug>MD 7400:2;DI<cr>
00000+R0 603E                        BRA.B      $00040+R0
00002+R0 4A7A8C2E                    TST.W      $00032(PC)
```

4.4 ENTERING AND MODIFYING SOURCE PROGRAMS

Entering or modifying program code in memory can be performed with the Memory Modify command, using the DI option. When this command is entered, 020Bug disassembles the instruction at each successive memory location, and replaces it with a new instruction assembled from the user's input.

4.4.1 Entering Source Code

The assembler is invoked using the ;DI option of the Memory Modify (MM) command, as shown here:

```
MM <ADDR>;DI
```

This will cause 020Bug to set the current memory location pointer to <ADDR> disassemble the instruction at that address, and prompt the user for input. The address must be an even value, so that instructions will be properly word aligned. Example:

```
020Bug>MM 4680;DI <cr>
00004680 00000000                    ORI.B      $00,D0 ?
```

If the user enters a valid source code statement, the assembler will process it, store the resulting object code at the current memory location, and rewrite that same line: the address is redisplayed, followed by the hex image of the new instruction, followed by the user's input line. Note that this is not a disassembly line. Example:

```
020Bug>MM 4680;DI <cr>
00004680 00000000                    ORI.B     $00,D0 ? CLR 120(A0)
(above line erased and rewritten as follows)
00004680 42680078                    CLR      120(A0)
00004684 FFFF                        DC.W      $FFFF ?
```

If the hardcopy printer is attached, the new line of code will be printed on the line below the original line.

After the input line is assembled and stored, the location pointer is moved up to the first byte after the end of the assembled instruction, and the process repeats, as shown above. Entering <cr> with no input line sequences to the next instruction. Example:

```
020Bug>MM 4684;DI <cr>
00004684 FFFF                        DC.W      $FFFF ? <cr>
00004686 FFFF                        DC.W      $FFFF ?
```

Entering . followed by <cr> terminates the command as usual. Example:

```
020Bug>MM 4684;DI <cr>
00004684 FFFF                        DC.W      $FFFF ? . <cr>
020Bug>
```

If the assembler cannot process the input line, the input line is redisplayed with a pointer to the part of the line the assembler could not process and the label "*** Unknown Field ***". Then the original line is displayed again. Example:

```
020Bug>MM 4680;DI <cr>
00010000 528B                        ADDQ.L    1,A3 ? lea.l 5(a0,d8),a4 <CR>
(line above overwritten)
00010000                             LEA       5(A0,D8),A4
-----------------------------------------------^
*** Unknown Field ***
00010000 528B                        ADDQ.L    #1,A3 ?
```

4.4.2 Entering Branch and Jump Addresses

When entering a source line containing a branch instruction (BRA, BGT, BEQ, etc) do not enter the offset to the branch's destination in the operand field of the instruction. Enter the destination address; the offset will be calculated by the assembler. The user must append the appropriate size extension to the branch instruction.

To refer to the current location in an operand expression the character "*" (asterisk) can be used. Examples:

```
00030000   60004094                  BRA *+$4096

00030000   60FE                      BRA.B *

00030000   4EF900030000              JMP *

00030000   4EF001300030000           JMP (*,A0,D0)
```

In the case of forward branches or jumps the absolute address of the destination may not be known as the program is being entered. The user may temporarily enter an "*" for branch to self in order to reserve space. After the actual address is discovered, the line containing the branch instruction can be re-entered using the correct value.

4.4.3 Inserting Additional Instructions

It may be neccessary for the user to go back and insert one or more additional instructions to a program already in memory. This can be done using the Block Move (BM) command to move part of the program up in memory, leaving a space where additional instructions can be placed. However, this will not alter any of the PC-relative addresses or branches in the program, and any such references which cross the gap must be corrected. Any reference which is not corrected can cause the program to fail.

4.5 ASSEMBLER OUTPUT/PROGRAM LISTINGS

A listing of the program can be obtained by using the Memory Display (MD) command with the ;DI option. The MD command expects a starting address and a line count in the command line. When the ;DI option is selected, the the line count is the number of instructions disassembled and displayed. If no line count is given the number defaults to 8.

A hard copy listing of the program can be obtained by using the Printer Attach (PA) command to activate the Port 1 printer before entering the MD command. This will produce a listing on the printer as well as on the console terminal.

Note again, that the listing may not correspond exactly to the program as entered.

SYSTEM CALLS

5.1 INTRODUCTION

This chapter describes the 020Bug TRAP #15 handler, which allows system calls from user programs. The system calls can be used to access selected functional routines contained within 020Bug, including input and output routines. Trap #15 may also be used to transfer control to 020Bug at the end of a user program (see the .RETURN function, section 5.2.17).

In the descriptions of some input and output functions, reference is made to the "default input port" or the "default output port". After power-up or reset, the default input and output port is initialized to be port 0 (the GMX Micro-20 terminal port). The defaults may be changed, however, using the .REDIR_I and .REDIR_O functions, as described in section 5.2.16.

5.1.1 Invoking System Calls Through TRAP #15

To invoke a system call from a user program simply insert a TRAP #15 instruction into the source program. The code corresponding to the particular system routine is specified in the word following the TRAP opcode, as shown in the following example.

Format in user program:

```
TRAP #15      System call to 020Bug
DC.W $xxxx    Routine being requested (xxxx = code)
```

In some of the examples shown in the following descriptions a SYSCALL macro is used. This macro simply does the TRAP #15 call followed by the Define Constant for the function code. For clarity, the SYSCALL macro is as follows:

```
SYSCALL       MACRO
TRAP          #15
DC.W          \1
ENDM
```

Using the SYSCALL macro, the system call would appear in the user program as follows:

```
SYSCALL       <routine name>
```

It is of course necessary to create an equate file with the routine names equated to their respective codes.

5.1.2 String Formats for I/O

Within the context of the TRAP #15 handler there are two formats for strings:

Pointer/Pointer Format - The string is defined by a pointer to the first character and a pointer to the last character + 1.

Pointer/Count Format - The string is defined by a pointer to a count byte which contains the count of characters in the string followed by the string itself.

A line is defined as a string followed by CRLF.

5.2 SYSTEM CALL ROUTINES

Table 5-1 summarizes the TRAP 15 functions. Refer to the write-ups on the utilities for specific use information.

TABLE 5-1. 020Bug SYSTEM CALL ROUTINES

Code	Function	Description	Page
0000	.INCHR	Input character	5-3
0001	.INSTAT	Input serial port status	5-4
0002	.INLN	Input line (pointer/pointer format)	5-5
0003	.READSTR	Input string (pointer/count format)	5-6
0004	.READLN	Input line (pointer/countformat)	5-7
0020	.OUTCHR	Output character	5-8
0021	.OUTSTR	Output string (pointer/pointer format)	5-9
0022	.OUTLN	Output line (pointer/pointer format)	5-9
0023	.WRITE	Output string (pointer/count format)	5-10
0024	.WRITELN	Output line (pointer/count format)	5-10
0025	.WRITDLN	Output line with data (pointer/count format)	5-11
0026	.PCRLF	Output carriage return and line feed	5-13
0027	.ERASLN	Erase line	5-14
0028	.WRITD	Output string with data (pointer/count format)	5-11
0050	.GETCLK	Get time data from clock	5-15
0051	.PUTCLK	Write time data to clock	5-16
0052	.OUTCLK	Output time data as a string	5-17
0060	.REDIR	Redirect I/O of a TRAP 15 function	5-18
0061	.REDIR_I	Redirect input	5-19
0062	.REDIR_O	Redirect output	5-19
0063	.RETURN	Return to 020Bug	5-20
0064	.BINDEC	Convert binary to decimal	5-21

TRAP FUNCTION: .INCHR - Input character routine-

CODE: $0000

DESCRIPTION: Will read a character from the default input port. The character is returned in the stack.

ENTRY CONDITIONS:

```
SP ==> Space for character <byte>
       Word fill           <byte>
```

EXIT CONDITIONS DIFFERENT FROM ENTRY:

```
SP ==> Character <byte>
       Word Fill <byte>
```

EXAMPLE:

```
SUBQ.L   #2,SP          Allocate space for result
SYSCALL  .INCHR         Call INCHR
MOVE.B   (SP)+,D0       Load character in D0
```

TRAP FUNCTION: .INSTAT - Input serial port status-

CODE: $0001

DESCRIPTION: INSTAT is used to check the status of the default input port. The condition codes are set to indicate the port status. Input, output, and BREAK status are checked and returned. The port hardware is checked for a received character, and any that are present are moved into the internal buffer, with handshaking as appropriate. If at this point the buffer is empty, the Z condition code is returned set. If the port hardware is NOT ready to accept a character for transmission, or an XOFF character has been received as a handshake code (and not cancelled by a subsequent XON character), the C condition code is returned set. If the BREAK detect flag in the port hardware is set, the N condition code is returned set.

ENTRY CONDITIONS:

No arguments or stack allocation required

EXIT CONDITIONS DIFFERENT FROM ENTRY:

Z = 1 if the receiver buffer is empty
C = 1 if the transmitter is not ready or waiting for XON
N = 1 if the BREAK flag is set

```
EXAMPLE:  LOOP    SYSCALL   .INSTAT        get status
                  BMI       EXIT           BREAK? exit loop
                  BEQ       NOIN           input?
                  SUBQ.L    #2,A7          if so
                  SYSCALL   .INCHR         read one character
                  MOVE.B    (SP)+,(A0)+    into program's input buffer
                  BRA       LOOP

          NOIN    BCS       NOOUT          ready for output?
                  MOVE.B    (A1)+,-(SP)    get char from output buffer
                  SYSCALL   .OUTCHR        and send it
                  BRA       LOOP           check for more
          EXIT    ....
```

TRAP FUNCTION: .INLN - Input line routine-

CODE: $0002

DESCRIPTION: Used to read a line from the default input port. The buffer size should be at least 256 bytes.

ENTRY CONDITIONS:

SP ==> Address of string buffer <long>

EXIT CONDITIONS DIFFERENT FROM ENTRY:

SP ==> Address of last character in the string+1 <long>

EXAMPLE: If A0 contains the address where the string is to go;

```
SUBQ.L   #4,A7          Allocate space for result
PEA      (A0)           Push pointer to destination
TRAP     #15            (May also invoke by SYSCALL
DC.W     2               macro ("SYSCALL
MOVE.L   (A7)+,A1       Retrieve address of last character+1
```

NOTES: A line is a string of characters terminated by <CR>. The maximum allowed size is 254 characters. The terminating <CR> is not included in the string. Control character processing as described in section 1.6, Terminal Input/Output Control, is in effect.

TRAP FUNCTION: .READSTR - Read string into variable-length buffer-

CODE: $0003

DESCRIPTION: READSTR is used to read a string of characters from the default input port into a buffer. On entry the first byte in the buffer indicates the maximum number of characters that can be placed in the buffer. The buffer size should at least be equal to that number+2. The maximum number of characters that can be placed in a buffer is 254 characters. On exit the count byte indicates the number of characters in the buffer. Input terminates when a <CR> is received. All characters will be echoed to the default output port. The <CR> will not be echoed.

ENTRY CONDITIONS:

SP ==> Address of input buffer <long>

EXIT CONDITIONS DIFFERENT FROM ENTRY:

SP ==> Top of stack
The count byte contains the number of bytes in the buffer.

EXAMPLE: If A0 contains the string buffer address;

```
PEA     (A0)        Push buffer address
TRAP    #15         (May also invoke by SYSCALL
DC.W    3            macro ("SYSCALL .READSTR")
```

NOTES: This routine allows the caller to dictate the maximum length of input to be less than 254 characters. If more than characters are entered, then the buffer input is truncated. Control character processing as described in section 1.6, Terminal Input/Output Control, is in effect.

TRAP FUNCTION: .READLN - Read Line to fixed-length buffer-

CODE: $0004

DESCRIPTION: READLN is used to read a string of characters from the default input port. Characters are echoed to the default output port. A string consists of a count byte followed the characters read from the input. The count byte indicates the number of characters in the input string, excluding <CR><LF>. A string may be up to 254 characters.

ENTRY CONDITIONS:

SP ==> Address of input buffer <long>

EXIT CONDITIONS DIFFERENT FROM ENTRY:

SP ==> Top of stack
The first byte in the buffer contains the number of bytes in the buffer.

EXAMPLE: If A0 points to a 256 byte buffer;

```
PEA       (A0)          Load buffer address
SYSCALL   .READSTR      and read a line from default input port
```

NOTES: The caller must allocate 256 bytes for a buffer. Input may up to 254 characters. CRLF is sent to default output following echo of input. Control character processing as described in section 1.6, Terminal Input/Output Control, is in effect.

TRAP FUNCTION: .OUTCHR - Output Character routine-

CODE: $0020

DESCRIPTION: This function will output a character to the default output port.

ENTRY CONDITIONS:

SP ==> Character <byte>
Word fill <byte> (Placed automatically by MPU)

EXIT CONDITIONS DIFFERENT FROM ENTRY:

SP ==> Top of stack
Character is sent to the default I/O port.

```
EXAMPLE:        MOVE.B   D0,-(SP)       Send character in D0
                SYSCALL  .OUTCHR        To default output port
```

5.2.7 .OUTSTR, .OUTLN FUNCTIONS

TRAP FUNCTIONS: .OUTSTR - Output string to default output port-
.OUTLN - Output string along with CR/LF-

CODES: $0021
$0022

DESCRIPTION: OUTSTR will output a string of characters to the default output port. OUTLN will output a string of characters followed by a <CR><LF> sequence.

ENTRY CONDITIONS:

```
SP ==> Address of first character   <long>
+4     Address of last character+1  <long>
```

EXIT CONDITIONS DIFFERENT FROM ENTRY:

SP ==> Top of stack

EXAMPLE: If A0 = start of string
A1 = end of string+1

```
MOVEM.L  A0/A1,-(SP)   Load pointers to string
SYSCALL  .OUTSTR       And print it
```

TRAP FUNCTIONS: .WRITE - Output string-
.WRITELN - Output string with CR/LF-

CODES: $0023
$0024

DESCRIPTION: These output functions are designed to output strings formatted with a count byte followed by the characters of the string. The user passes the starting address of the string. The output goes to the default output port.

ENTRY CONDITIONS:

Four bytes of parameter positioned in stack as follow:

SP ==> Address of string <long>

EXIT CONDITIONS DIFFERENT FROM ENTRY:

SP ==> Top of stack

EXAMPLE: For example, the following section of code .....

```
MESSAGE1  DC.B     9, 'MOTOROLA'
MESSAGE2  DC.B     9, 'QUALITY!'
          PEA      MESSAGE1(PC)  Push address of string
          SYSCALL  .WRITE        Use TRAP #15 macro
          PEA      MESSAGE2(PC)  Push address of other string
          SYSCALL  .WRITE        Invoke function again
```

... would print out the following message:

MOTOROLA QUALITY !

Using function .WRITELN, however, instead of function .WRITE would output the following message:

MOTOROLA
QUALITY !

NOTES: The string must be formatted such that the first byte (the byte pointed to by the passed address) contains the count (in bytes) of the string.

TRAP FUNCTIONS: .WRITDLN - Output string with data and CR/LF-
.WRITD - Output string with data-

CODES: $0025
$0028

DESCRIPTION: These trap functions takes advantage of the monitor I/O routine which outputs a user string which has embedded variable fields in it. The user passes the starting address of the string and the address of a data stack from whence the data which will be inserted into the string will be read. The output goes to the default output port.

ENTRY CONDITIONS:

Eight bytes of parameter positioned in stack as follows:

SP ==> Address of string <long>
Data list pointer <long>

A separate data stack or data list arranged as follows:

Data list pointer => Data for 1st variable in string <long>
Data for next variable <long>
Data for next variable <long>

EXIT CONDITIONS DIFFERENT FROM ENTRY:

SP ==> Top of stack

EXAMPLE: For example, the following section of code .....

```
ERRMESSG  DC.B     $14, 'ERROR CODE = ¦10,8Z¦'

          MOVE.L   #3,-(A5)       Push error code on data stack
          PEA      (A5)           Push data stack location
          PEA      ERRMESSG(PC)   Push address of string
          SYSCALL  .WRITDLN       Invoke function
          TST.L    (A5)+          Deallocate data from data stack
```

... would print out the following message:

ERROR CODE = 3

NOTES: 1) The string must be formatted such that the first byte (the byte pointed to by the passed address) contains the count (in bytes) of the string (including the data field specifiers, described in #2 below).

2) Any data fields within the string must be represented as follows: "¦<radix>,<fieldwidth>[Z]¦" where <radix> is the base that the data is to be displayed in (in hexadecimal, i.e., "A" is base 10, "10" is base 16, etc.) and <fieldwidth> is the number of characters this data is to occupy in the output. The data is right justified and

left-most characters are removed to make the data fit. If the "Z" is included, leading zeroes will be suppressed in the output, except that one zero will always be printed for a zero value.

3) All data is to be placed in the stack as longwords. Each time a data field is encountered in the user string a longword will be read from the data stack to be displayed.

4) The data stack is not destroyed by this routine. If it is necessary that the space in the data stack be deallocated then this must be done by the calling routine, as shown in the above example.

5.2.10 .PCRLF FUNCTION .PCRLF

TRAP FUNCTION: .PCRLF - Print <CR><LF> sequence

CODE: $0026

DESCRIPTION: PCRLF will send a <CR><LF> sequence to the default output port.

ENTRY CONDITIONS:

No arguments or stack allocation required.

EXIT CONDITIONS DIFFERENT FROM ENTRY:

None

```
EXAMPLE:        SYSCALL  .PCRLF         Output CRLF
```

TRAP FUNCTION: .ERASLN - Erase line-

CODE: $0027

DESCRIPTION: Erase line is used to erase the line at the present cursor position. If the terminal type flag is set for hardcopy mode a <CR> <LF> is issued instead.

ENTRY CONDITIONS:

No arguments required.

EXIT CONDITIONS DIFFERENT FROM ENTRY:

The cursor is position at the beginning of a blank line.

EXAMPLE:

```
SYSCALL .ERASLN
```

TRAP FUNCTION: .GETCLK - Get clock contents -

CODE: $0050

DESCRIPTION: .GETCLK is used to read time and date data from the GMX Micro-20's hardware time-of-day clock.

ENTRY CONDITIONS:

SP ==> space for time data <8 bytes>

EXIT CONDITIONS DIFFERENT FROM ENTRY:

```
SP ==>  current second  (00-59) <byte>
        current minute  (00-59) <byte>
        current hour    (00-23) <byte>
        current weekday (01-07) <byte>
        current day     (01-31) <byte>
        current month   (01-12) <byte>
        current year    (00-99) <byte>
        padding         (00)    <byte>
```

```
EXAMPLE:    SUBQ.L   #8,SP          Allocate space for result
            SYSCALL  .GETCLK        Call GETCLK
```

NOTES: This function reads the clock twice to insure that the value is stable. If the value read is not the same each time, then the clock is read again. If after five tries the clock has not returned the same value, all 00s is returned to the calling routine. .GETCLK assumes the clock is in 24-hour mode.

TRAP FUNCTION: .PUTCLK - Get clock contents -

CODE: $0051

DESCRIPTION: .PUTCLK is used to write time and date data into the GMX Micro-20's hardware time-of-day clock.

ENTRY CONDITIONS:

```
SP ==>  current second  (00-59) <byte>
        current minute  (00-59) <byte>
        current hour    (00-23) <byte>
        current weekday (01-07) <byte>
        current day     (01-31) <byte>
        current month   (01-12) <byte>
        current year    (00-99) <byte>
        padding         (00)    <byte>
```

EXIT CONDITIONS DIFFERENT FROM ENTRY:

SP ==> Top of stack

EXAMPLE:

```
MOVE.L   TIMELO,-(SP)   put time data on the stack
MOVE.L   TIMEHI,-(SP)
SYSCALL  .PUTCLK        Call PUTCLK
```

NOTES: This function writes the clock with the data given regardless of its value. Nonsense values can cause the clock to malfunction.

TRAP FUNCTION: .OUTCLK - Output clock data -

CODE: $0052

DESCRIPTION: .OUTCLK is used to output a set of time and date data to the default I/O path.

ENTRY CONDITIONS:

SP ==> address of clock data block

```
data pointer ==>  current second  (00-59) <byte>
                  current minute  (00-59) <byte>
                  current hour    (00-23) <byte>
                  current weekday (00-06) <byte>
                  current day     (01-31) <byte>
                  current month   (01-12) <byte>
                  current year    (00-99) <byte>
                  padding         (00)    <byte>
```

EXIT CONDITIONS DIFFERENT FROM ENTRY:

SP ==> Top of stack

EXAMPLE: if A0 points to a time data block....

```
        PEA      (A0)
        SYSCALL  .OUTCLK        Call OUTCLK
```

produces a string in the form

WEDNESDAY 01/02/99 23:34:56

NOTES: This function outputs a time and date string followed by a CR/LF. The data source could be either the GMX Micro-20 hardware clock, read using the .GETCLK function, or a program or data file. If the weekday given is not from 0 to 6, "BAD WEEK DAY" will be displayed instead of the day name.

TRAP FUNCTION: .REDIR - Redirect I/O function-

CODE: $0060

DESCRIPTION: .REDIR is used to select an I/O port and at the same time invoke a particular I/O function. The invoked I/O function will read or write to the selected port.

ENTRY CONDITIONS:

```
SP ==> Port                          <word>
       I/O function to call          <word>
       Parameters of I/O function    <size specified by function>
       Space for results             <size specified by function>
```

EXIT CONDITIONS DIFFERENT FROM ENTRY:

```
SP ==> Result                        <size specified by function>
```

EXAMPLE: ---

NOTES: To use .REDIR, the caller should first allocate space and push the parameters required by the desired I/O function in the stack:

```
SUBQ.L   #2,A7          Allocate space (no parameters required by
                          .INCHR)
```

Then the parameters required by .REDIR should be pushed and a call is made to .REDIR:

```
MOVE.W   #.INCHR,-(SP) Load function code
MOVE.W   #1,-(SP)      Load port number
SYSCALL  .REDIR        Redirect I/O function
```

Finally, the results are popped from the stack:

```
MOVE.B   (SP)+,D0       Read character
```

The above example reads a character from port 1 using .REDIR.

TRAP FUNCTIONS: .REDIR_I - Redirect input-
.REDIR_O - Redirect output-

CODES: $0061
$0062

DESCRIPTION: The .REDIR_I and .REDIR_O functions are used to change the default port number of the input and output ports, respectively. This is a permanent change, that is, it will remain in effect until a new .REDIR command is issued.

ENTRY CONDITIONS:

SP ==> Port Number <word>

EXIT CONDITIONS DIFFERENT FROM ENTRY:

SP ==> Top of stack
.SIO_IN - Loaded with a new mask if .REDIR_I called
.SIO_OUT - Loaded with a new mask if .REDIR_ O called

```
EXAMPLE:        MOVE.W   #1,-(SP)      Load port number
                SYSCALL  .REDIR_I      Set it as new default
```

TRAP FUNCTION: .RETURN - Return to 020Bug-

CODE: $0063

DESCRIPTION: .RETURN is used to return control to 020Bug from the target program in an orderly manner. First any breakpoints inserted in the target code are removed. Then the target state is saved in the register image area. Finally the routine returns to 020Bug.

ENTRY CONDITIONS:

No arguments required.

EXIT CONDITIONS DIFFERENT FROM ENTRY:

Control is returned to 020Bug.

EXAMPLE:

```
SYSCALL  .RETURN        Return to 020Bug
```

TRAP FUNCTION: .BINDEC FUNCTION (Used to calculate the BCD equivalent of the binary number specified)

CODE: $0064

DESCRIPTION: .BINDEC takes a 32-bit unsigned binary number and changes it to an equivalent BCD (Binary Coded Decimal Number).

ie. $30 ⟶ 48

ENTRY CONDITIONS:

```
SP ==> Argument:Hex number<long>
       Space for result   <2 long>
```

EXIT CONDITIONS DIFFERENT FROM ENTRY:

```
SP ==> Decimal number (2 MS DIGITS) <long>
                      (8 MS DIGITS) <long>
```

EXAMPLE:

```
SUBQ.L   #8,A7          Allocate space for result
MOVE.L   D0,-(SP)       Load hex number
SYSCALL  .BINDEC        Call .BINDEC
MOVEM.L  (SP)+,D1/D2    Load result
```

APPENDIX A

ALTERNATE ROM PROGRAMS

The GMX Micro-20 has four EPROM sockets, which can hold up to 256 Kbytes of EPROM. 020Bug is installed in the first part of this memory. The remaining memory may be used for many different functions. For example, the bootstrap loader for a disk operating system could be placed there. This is done by GMX if an operating system was purchased with the GMX Micro-20. The user may also place his own programs in the ROM.

Available ROM

The address space of the ROM is $00800000 to $0083FFFF, allowing 256 Kbytes of ROM. However, the board is shipped with four 32K x 8 devices, which provide only 128 Kbytes of ROM. Version 2.5 of GMX 020Bug occupies the first 88 Kbytes, so a user program should start at $816000 or higher to avoid conflicts with 020Bug. If a bootstrap loader is installed, it will begin at $816000; its size will depend on the operating system purchased, and can be determined by examining the ROM with the MD command. A user program can be safely installed in the ROM above the boot area.

Adding More ROM

The EPROM sockets on the GMX Micro-20 can accept four 64K x 8 EPROMs for 256 Kbytes of ROM. A user who needs more ROM should replace the supplied parts with larger parts. Refer to the Hardware Technical Manual section on "ROM Sockets" for details of this conversion.

ROM Data Organization

The ROM is organized as 64K longwords of 32 bits each, so that a longword read from ROM fetches one byte from each device. Therefore, any information (programs or data) stored in the ROM must be "split up" among the four EPROMs. The first byte of each longword goes in EPROM #1 (referred to as part U-13 in the Hardware Technical Manual), the second in EPROM #2 (U-10), the third in EPROM #3 (U-8), and the fourth in EPROM #4 (U-6). The diagram below shows how the longwords $00001FFC and $0080079C would be stored at $800000.

Address	EPROM #1 (U-13)	EPROM #2 (U-10)	EPROM #3 (U-8)	EPROM #4 (U-6)
$800000	$00	$00	$1F	$FC
$800004	$00	$80	$07	$9C

Automatic Invocation Of A ROM Program

There are several ways to invoke a program in the ROM. It can be called by a user program in RAM, or started from 020Bug with the "GO" command. A particular ROM program can be selected to be started by the "OS" command. This last method is accomplished by using the Operating System vector in 020Bug.

At address $00014 in the ROM (absolute address $800014), there is a four-byte vector which contains the address of an eight-byte block of data. This block contains the starting address and initial stack pointer value for a program in ROM, as shown below. If the user enters the 020Bug command "OS", 020Bug uses the vector at $800014 to fetch a stack pointer value and execution address, and jumps to the specified address.

Address	Contents	
$800014	$816000	points to $816000
...	...	
...	...	
...	...	
$816000	$003FFC	new stack pointer value
$816004	$817000	starting address of user program
$817000	xxx	first instruction of user program

This process can be invoked automatically at power-up/Reset, by setting switch S1-1 to the "OFF" position. Refer to Section 6 of the Hardware Setup Manual, "DIP-SWITCH OPTIONS".

NOTE: The hardware confidence test is always performed at power-up/Reset. Even if switch S1-1 is OFF, the confidence test is performed before the user program starts. If the system does not pass the confidence test, the user program will not start. Also, no useful initialization of any system hardware is performed; the user program must do all needed initialization for itself.

APPENDIX C

S-RECORD OUTPUT FORMAT

The S-record format for output modules was devised for the purpose of encoding programs or data files in a printable format for transportation between computer systems. The transportation process can thus be visually monitored and the S-records can be more easily edited.

S-RECORD CONTENT

When viewed by the user, S-records are essentially character strings made of several fields which identify the record type, record length, memory address, code/data and checksum. Each byte of binary data is encoded as a 2-character hexadecimal number: the first character representing the high-order 4 bits, and the second the low-order 4 bits of the byte.

The 5 fields which comprise an S-record are shown below:

type record length address code/data checksum

where the fields are composed as follows:

Field	Printable Characters	Contents
type	2	S-records type -- S0, S1, etc.
record length	2	The count of the character pairs in the record, excluding the type and the record length.
address	4, 6, or 8	The 2-, 3-, or 4-byte address at which the data field is to be loaded into memory.
code/data	0-2n	From 0 to n bytes of executable code, memory-loadable code, or descriptive informantion. For compatibility with teletypewriters, some programs may limit the number of bytes to as few as 28 (56 printable characters in the S-record).
checksum	2	The least significant byte of the one's complement of the sum of the values represented by the pairs of characters making up the records length, address, and the code/data fields.

Each records may be terminated with a CR/LF/NULL. Additionally, an S-record may have an initial field to accommodate other data such as line numbers generated by some time-sharing systems.

Accuracy of transmission is ensured by the record length (byte count) and checksum fields.

S-RECORD TYPES

Eight types of S-records have been defined to accommodate the several needs of the encoding, transportation and decoding functions. The various Motorola upload, download and other records transportation control programs, as well as cross assemblers, linkers and other file-creating or debugging programs, utilize only those S-records which serve the purpose of the program. For specific information on which S-records are supported by a particular program, the user's manual for the program must be consulted. 020Bug supports S0, S1, S2, S3, S7, S8, and S9 records.

An S-record format module may contain S-records of the following types:

S0 The header record for each block of S-records. The code data field may contain any descriptive information identifying the following block of S-records. Under VERSAdos, the resident linker's IDENT command can be used to designate module name, version number, revision number, and description information which will make up the header records. The address field is normally zeroes.

S1 A record containing code/data and the 2-byte address at which the code/data is to reside.

S2 A record containing code/data and the 3-byte address at which the code/data is to reside.

S3 A record containing code/data and the 4-byte address at which the code/data is to reside.

S5 A record containing the number of S1, S2 and S3 records transmittted in a particular block. This count appears in the address field. There is no code/data field.

S7 A termination record for a block of S3 records. The address field may optionally contain the 4-byte address of the instruction to which control is to be passed. There is no code/data field.

S8 A termination record for a block of S2 records. The address field may optionally contain the 3-byte address of the instruction to which control is to be passed. There is no code/data field.

S9 A termination record for a block of S1 records. The address field may optionally contain the 2-byte address of the instruction to which control is to be passed. Under VERSAdos, the resident linker's ENTRY command can be used to specify this address. If not specified, the first entry point specification encountered in the object module input will be used. There is no code/data field.

Only one termination record is used for each block of S-records. S7 and S8 records are usually used only when control is to be passed to a 3- or 4-byte address. Normally, only one header record is used, although it is possible for multiple header records to occur.

CREATION OF S-RECORDS

S-record-format programs may be producted by several dump utilities, debuggers, VERSAdos' resident linkage editor, or several cross assemblers or cross linkers. On VERSAdos, the Build Load Module (MBLM) utility allows an executable load module to be build from S-records, and has a counterpart utility in BUILDS, which allows an S-records file to be created from a load module. Several programs are available for downloading a file in S-records format from a host system to an 8-bit microprocessor-based or a 16-bit microrocessor-based system. Programs are also available for uploading an S-records file to or from an EXORmacs system.

EXAMPLE

Shown below is a typical S-record-format module, as printed or displayed:

```
S006000048445218
S1130000285F245F2212226A000424290008237C2A
S1130010000200080008262900185381234100181 3
S113002041E900084E42234300182342000824A952
S107003000144ED492
S9030000FC
```

The module consists of one S0 record, four S1 records, and an S9 record.

The S0 record is comprised of the following character pairs:

S0 S-record type S0, indicating that it is a header record.

06 Hexadecimal 06 (decimal 6), indicating that six character pairs (or ASCII bytes) follow.

00
00 Four-character 2-byte address field, zeroes in this example.

48
44 ASCII H, D and R - "HDR".
52

1B The checksum.

The first S1 record is explained as follows:

S1 S-record type S1, indicating that it is a code/data record to be loaded/ verified at a 2-byte address.

13 Hexadecimal 13 (decimal 19), indicating that 19 character pairs, representing 19 bytes of binary data, follow.

00 Four-character 2-byte address field; hexadecimal address 0000, where the 00 data which follows is to be loaded.

The next 16 character pairs of the first S1 record are the ASCII bytes of the actual program code/data. In this assembly language example, the hexadecimal opcodes of the program are written in sequence in the code/data fields of the S1 records:

Opcode	Instruction	
285F	MOVE.L	(A7)+,A4
245F	MOVE.L	(A7)+,A2
2212	MOVE.L	(A2),D1
226A0004	MOVE.L	4(A2),A1
24290008	MOVE.L	FUNCTION(A1),D2
237C	MOVE.L	00RCEFUNC,FUNCTION(A1)

(The balance of this code is continued in the code/data fields of the remaining S1 records and stored in memory location 0010, etc.)

2A The checksum of the first S1 record.

The second and third S1 records each also contain $13 (19) character pairs and are ended with checksums 13 and 52, respectively. The fourth S1 record contains 07 character pairs and has a checksum of 92.

The S9 record is explained as follows:

S9 S-record type S9, indicating that it is a termination record.

03 Hexadecimal 03, indicating that three character pairs (3 bytes) follow.

00
00 The address field, zeroes.

FC The checksum of the S9 record.

Each printable character in an S-record is encoded in hexadecimal (ASCII in this example) representation of the binary bits which are actually transmited. For example, the first S1 record above is sent as:

Type				Length				Address							
S		1		1		3		0		0		0		0	
5	3	3	1	3	1	3	3	3	0	3	0	3	0	3	0
0101	0011	0011	0001	0011	0001	0011	0011	0011	0000	0011	0000	0011	0000	0011	0000

Code/Data									Checksum			
2		8		5		F			2		A	
3	2	3	8	3	5	4	6	....	3	2	4	1
0011	0010	0011	1000	0011	0101	0100	0110	....	0011	0010	0100	0001

APPENDIX D

020Bug DIAGNOSTIC FIRMWARE GUIDE

TABLE OF CONTENTS

TABLE OF CONTENTS (cont'd)

TABLE OF CONTENTS (cont'd)

1.0 SCOPE

This diagnostic guide contains information about the operation and use of the GMX Micro-20 Diagnostic Firmware Package, hereafter referred to as "the diagnostics". Sections 3 and 4 give the user guidance in setting up the system and invoking the various utilities and tests. Section 5 describes utilities available to the user. Sections 6, 7, 8, 9, 10, 11, and 12 are guides to using each test.

2.0 OVERVIEW OF DIAGNOSTIC FIRMWARE

The GMX Micro-20 diagnostic firmware package consists of the 020Bug monitor plus a battery of utilities and test modules for exercising, testing, and debugging the GMX Micro-20's hardware.

The diagnostics are menu-driven for ease of use. The "HE" command (explained fully in section 5.1) displays a menu of all available diagnostic functions (the tests and utilities). Each test has a subtest menu which may be called using the "HE" command.

3.0 SYSTEM START-UP

When the system is turned on, a short "confidence test" is performed to insure that the MPU, ROMs, address decoders, and RAM memory are all working at a minimum level. If any defects in the system are detected at this time, it is possible that the system will not work well enough even to do console output. So 020BUG does not "come up"; instead an error code is output in the form of coded flashes of LED 2. The code is four bits long, with 1s represented by steady pulses of the LED and 0s by flickering pulses. Bits are separated by short pauses. Thus error code 1001 would be represented by

Steady - Flicker - Flicker - Steady

After the code is completed, there is a longer pause, then the code is repeated. This will continue until the system is restarted. Each code indicates a different hardware problem as shown in the table below.

Confidence Test Error Codes

0000	0	Not used
0001	1	68020 register error
0010	2	68020 instruction error
0011	3	Reset failure
0100	4	PROM checksum error
0101	5	Addressing mode error
0110	6	Exception failed to occur
0111	7	Wrong exception generated
1000	8	Stuck interrupt mask bit in 68020 Status Register
1001	9	Unexpected interrupt
1010	10	Memory error
1011	11	Unexpected bus error
1100	12	Bus error in accessing DUARTs
1101	13	Partial longword write error
1110	14	Console port output inhibited

4.0 DIAGNOSTIC MONITOR

The test described herein are called via a common diagnostic monitor, hereafter called "monitor". This monitor is menu driven and provides input/output facilities, command parsing, error reporting, interrupt handling, and a multi-level directory. A complete description of the capabilities and structure of the monitor can be found in the GMX Micro-20 diagnostic monitor specification.

4.1 Monitor Start-Up

When the monitor is first brought up, following power up or reset, the following text should be displayed on the console video display terminal:

```
GMX Micro-20 Debugger/Diagnostics Version 2.4 - 05/19/86
FRIDAY   09/24/86   12:00:08
020Bug>
```

To switch to the diagnostics directory, enter "SD"; the prompt will change to "M20Diag>". "SD" is explained in detail in section 4.6.

4.2 Command Entry and Directories

Entry of commands is made when the prompt "M20Diag>_ " appears. The name of the command is entered before pressing "RETURN". Multiple commands may be entered. If a command expects parameters and another command is to follow it, separate the two with an exclamation mark "!". For instance, to invoke the commands MT A and MT B, the command line would read "MT A ! MT B". Spaces are not required but are shown here for legibility.

Most commands consist of a command name that is listed in a main (root) directory and a subcommand that is listed in the directory for that particular command. In the main directory are commands like "MH" and "MT". These commands are used to refer to a set of lower level commands.

To call up a particular bus test, one would enter (on the same line) "CA20 A". This command would cause the monitor to find the "CA20" subdirectory, and then to execute the command "A" from that subdirectory.

The diagram shown below is provided to illustrate the directory structure.

Single Level Commands	HE	Help
	LE	Loop on Error
Two Level Commands	CA20 F	On-chip cache
		F Basic caching
	MH A	Miscellaneous Hardware
		A FPC instructions

4.3 HELP - Command "HE"

Online documentation has been provided in the form of a "HELP" command (syntax: "HE"). This command will display the main menu if no parameters are entered, or a menu of each subdirectory if the name of that subdirectory is entered. For example, to bring up a menu of all the on-chip cache tests, one would enter "HE CA20". When a menu is too long to fit on the screen, it will pause until the operator depresses "RETURN".

4.4 SELF-TEST - Prefix "ST"

The monitor provides an automated test mechanism called "Self Test". Prefixing a command line with "ST" will cause the monitor to run the tests specified if they are included in the internal self test directory. Entering "ST" with no parameters will run most of the GMX Micro-20 diagnostics. To run most of the memory tests, one would enter "ST MT". Entering "ST CA20 G CA20 H" would cause the monitor to run the tests CA20 G and CA20 H.

Each test that is not included in the self test chain for that particular command is listed in the section pertaining to the command (i.e., see section 10 for the MT commands that are not in the self test chain).

4.5 SELF-TEST LED - Prefix "STL"

This command is identical to the ST command, except for one special feature. When the set of specified test modules invoked with "STL" is completed, a test for errors is made. If any of the tests returned an error, the MPU enters a loop which blinks the Status LED (LED 2) rapidly and continuously until a RESET occurs or BREAK is received on the console.

This command allows the user to start a test or set of tests, then disconnect the console terminal for other uses. If the "LC" command is used, the test or tests will be repeated until an error occurs or the user intervenes. If the blinking LED signals that an error has been is detected, then the user can attach a terminal, enter BREAK to resume normal command entry, and use the "DE" command to display the error table, and find out which test failed.

If switch S1-2 is ON, then at power-on/reset the "LC" and "STL" commands will be invoked immediately, causing the system to execute the full self-test command set in this mode, and no terminal is required at all.

4.6 SWITCH DIRECTORIES - Command "SD"

The debugger is the root or main directory after power-on/reset. At this point, only the commands for 020Bug will function. Another directory is maintained for the diagnostic commands. To access the diagnostic directory (and enable the diagnostic tests), enter "SD". The diagnostic commands are now available in addition to the debugger commands. To return to the debugger directory, the command "SD" is entered again. When the 020Bug directory is selected, the prompt will read "020Bug>". When the diagnostic directory is selected, the prompt will read "M20Diag>". The purpose of this feature is to allow the end user to access 020Bug without the diagnostics being visible.

4.7 DISPLAY PASS COUNT - Command "DP"

When Loop Continuous mode is used (see below), a counter is incremented for each pass through the specified test or tests. This counter is preserved after the test loop is exited by BREAK or error, and can be displayed by using the DP command.

4.8 ZERO PASS COUNT - Command "ZP"

The pass counter is automatically set to zero at power-on/reset, but is never reset otherwise, except by this command. Using this command before beginning a test run insures that the pass count reflects only passes in the current run, and not in a previous run. However, if the user needs to interrupt and restart the test run, the pass count is preserved.

4.9 DISPLAY ERROR COUNTERS - Command "DE"

Each test or command in the diagnostic monitor has an individual error counter. As errors are encountered in a particular test, that error counter will be incremented. If one were to run a self test or just a series of tests, the results could be broken down as to which tests passed by examining the error counters. "DE" will display the results of a particular test if the name of that test follows "DE".

4.10 ZERO ERROR COUNTERS - Command "ZE"

The error counters are initialized to zero at power-on/reset, but it is occasionally desirable to reset them to zero at a later time. This command will reset all of the error counters to zero. The error counters can be individually reset by entering the specific test name following the command. Example: ZE MH A will clear the error counter associated with MH A.

4.11 NON-VERBOSE MODE - Prefix "NV"

The tests included for the GMX Micro-20 will frequently display a substantial amount of data upon detecting an error. To avoid the necessity of watching the scrolling display, a mode is provided that supresses all messages except "PASSED" or "FAILED". This mode is called "nonverbose" and is invoked prior to calling a command by entering "NV". "NV ST MT" would cause the monitor to run the memory self test, but only show the names of the subtests and the results (pass/fail).

4.12 LOOP CONTINUOUS MODE - Prefix "LC"

To endless repeat a test or series of tests, the prefix "LC" is entered. This loop will include everything on the command line with the "LC". To break the loop, depress "BREAK". Certain tests disable the "BREAK" key interrupt, so depressing the reset button may become necessary.

4.13 LOOP ON ERROR MODE - Prefix "LE"

Occasionally, when an oscilloscope or logic analyzer is in use, it becomes desirable to repeat a test at the point where an error is detected. "LE" accomplishes that for most of the tests. To invoke "LE", enter it before the test that is to run in "loop on error" mode on the same command line.

4.14 STOP ON ERROR MODE - Prefix "SE"

This command sets a flag which halts testing as soon as any test reports an error. If this command is entered on a command line before a selftest command or any other, testing will stop at the first error detected. This prevents error information from scrolling off the screen. It is compatible with Loop Continuous mode, but not with Loop on Error.

5.0 UTILITIES

The monitor is supplemented by two utilities that are separate and distinct from the monitor itself and the diagnostics.

5.1 WRITE LOOP - Command "WL.size"

The "WL.size" command causes a repeated write of the specified size to the specified memory location. This command is intended as a technician aid for debug once specific fault areas are identified. The write loop is very short in execution so that measuring devices such as oscilloscopes may be utilized in tracking failures.

The size of the command may be specified as B for byte, W for word, or L for longword.

The command requires two parameters: target address and data to be written. The address and data are both hexadecimal values and must be preceded by a 0 if the first digit is other than 0-9, i.e., $FF would be entered as "0FF". To write $00 out to address $FFFB0030, one would enter "WL.B 0FFFB0030 00". Omission of either or both parameters will cause prompting for the missing values.

This command is set up to continue writing even if the write produces a Bus Error Exception, so writing to an address which fails to respond is possible.

5.2 READ LOOP - Command "RL.size"

The RL.size command causes a repeated read of specified size from the specified memory location. This command is intended as a technician aid for debug once specific fault areas are identified. The read loop is very short in execution so that measuring devices such as oscilloscopes may be utilized in tracking failures.

The size of the command may be specified as B for byte, W for word, or L for longword.

The command requires one parameter: target address. The address is a hexadecimal value and must be preceded by a 0 if the first digit is other than 0-9, i.e., $FFFB0030 would be entered as "0FFFB0030". To read from address $FFFB0030, one would enter "RL.B 0FFFB0030". Omission of the parameter will cause prompting for the missing value.

This command is set up to continue reading even if the read produces a Bus Error Exception, so reading from an address which fails to respond is possible.

5.3 DISPLAY SWITCHES - Command "DS"

This command reads the five-position DIP switch (SW1) on the GMX Micro-20 and displays its value in binary, where 0 represents a switch in the ON position and 1 represents a switch in the OFF position. It repeats this display on the same line indefinitely, until terminated by BREAK. If the user changes a switch, the display will be updated immediately.

5.4 DISPLAY JUMPERS - Command "DJ"

This command reads the baud rate jumper block (JAx) and displays the setting as two binary digits, where 0 represents an installed jumper and 1 represents a missing jumper. It repeats this display on the same line indefinitely, until terminated by BREAK. If the user changes a jumper, the display will be updated immediately.

6.0 RAM TESTS

6.1 General Description

This set of tests exercises the GMX Micro-20's on-board RAM, or any selected segment of the RAM.

TABLE T-1. MEMORY DIAGNOSTIC TESTS

Monitor Command	Title	Section	Page
MT A	Set Function Code	6.2	D-8
MT B	Set Starting Address	6.3	D-9
MT C	Set Stop Address	6.4	D-10
MT D	Random Inversion Test	6.5	D-11
MT E	March Address Test	6.6	D-12
MT F	Walk A Bit Test	6.7	D-13
MT G	Refresh Test	6.8	D-14
MT H	Random Byte Test	6.9	D-15
MT I	Program Test	6.10	D-16
MT J	TAS Test	6.11	D-17
MT K	Test 0000-1FFF	6.12	D-18
MT L	Partial Longword Writes Test	6.13	D-20

6.2 Set Function Code MT A

6.2.1 Description

This command allows the user to select the function code used in most of the memory tests. The exceptions to this are "TAS test" and "program test".

6.2.2 Command Input

M20Diag>MT A [new value] <CR>

6.2.3 Response/Messages

If the user supplied the optional new value, then the display will appear as follows:

```
M20Diag>MT A [new value] FUNCTION CODE=<new value>
M20diag>
```

If a new value was not specified by the user, then the old value will be displayed and the user will be prompted for a new value.

```
M20Diag>MT A FUNCTION CODE=<current value> ?[new value]<CR>
FUNCTION CODE=<new value>
M20Diag>
```

This command may be used to display the current value without changing it by depressing "RETURN" without entering the new value.

```
M20Diag>MT A FUNCTION CODE=<current value> ?<CR>
FUNCTION CODE=<current value>
M20Diag>
```

6.3 Set Starting Address MT B

6.3.1 Description

This command allows the user to select the start address used by all of the memory tests. The default starting address (set at power-on/reset) is $2000. Memory below this address is used for monitor variables and for the monitor stack. Use the MT K command to test this memory.

6.3.2 Command Input

```
M20Diag>MT B [new value] <CR>
```

6.3.3 Response/Messages

If the user supplied the optional new value, then the display will appear as follows:

```
M20Diag>MT B [new value]
Start Addr=<new value>
M20Diag>
```

If a new value was not specified by the user, then the old value will be displayed and the user will be prompted for a new value.

```
M20Diag>MT B
Start Addr=<current value> ?[new value]<CR>
Start Addr=<new value>
M20Diag>
```

This command may be used to display the current value without changing it by depressing "RETURN" without entering the new value.

```
M20Diag>MT B
Start Addr=<current value> ?<CR>
Start Addr=<current value>
M20Diag>
```

Note: If a new value is specified, it will be truncated to a longword boundary and, if greater than the value of the stop address, will replace the stop address. The start address is never allowed to be higher in memory than the stop address. These changes will occur before another command will be processed by the monitor.

6.4 Set Stop Address MT C

6.4.1 Description

This command allows the user to select the stop address used by all of the memory tests. If the user enters a value less than the current start address, it will be replaced by the start address. The stop address is also truncated to a longword boundary.

6.4.2 Command Input

```
M20Diag>MT C [new value] <CR>
```

6.4.3 Response/Messages

If the user supplied the optional new value, then the display will appear as follows:

```
M20Diag>MT C [new value]
Stop Addr=<new value>
M20Diag>
```

If a new value was not specified by the user, then the old value will be displayed and the user will be prompted for a new value.

```
M20Diag>MT C
Stop Addr=<current value> ?[new value]<CR>
Stop Addr=<new value>
M20Diag>
```

This command may be used to display the current value without changing it by depressing "RETURN" without entering the new value.

```
M20Diag>MT C
Stop Addr=<current value> ?<CR>
Stop Addr=<current value>
M20Diag>
```

Note: If a new value is specified, it will be truncated to a longword boundary and, if less than the value of the starting address, will be replaced by the starting address. The stop address is never allowed to be lower in memory than the starting address. These changes will occur before another command will be processed by the monitor.

6.5 Random Inversion Test MT D

6.5.1 Description

This test performs an inversion test with pseudo-random data from STARTING ADDRESS to STOP ADDRESS. It is implemented as follows:

Step 1. The time-of-day clock is read to provide a seed for the pseudo-random sequence.

Step 2. Each longword from START to STOP is written with a value in the pseudo-random sequence and then with its inverse.

Step 3. Each longword from STOP to START is checked for the inverse of its pseudo-random value, and then written with the value itself.

Step 4. Each longword from START to STOP is checked for its pseudo-random value.

6.5.2 Command Input

M20Diag>MT D <CR>

6.5.3 Response/Messages

After the command is entered, the display should appear as follows:

D Random Inversion Test ..... Running --->

If an error is encountered, then the memory location and other related information will be displayed (see section 6.13).

D Random Inversion Test ..... Running --->

(error-related information) ..... FAILED

If no errors are encountered, then the display will appear as follows:

D Random Inversion Test ..... Running ---> PASSED

6.7 March Address Test MT E

6.6.1 Description

This command performs a march address test from STARTING ADDRESS to STOP ADDRESS.

The march address test has been implemented in the following manner:

Step 1. All memory locations from STARTING ADDRESS up to STOP ADDRESS are cleared to 0.

Step 2. Beginning at STOP ADDRESS and proceeding downward to STARTING ADDRESS, each memory location is checked for bits that did not clear, and then written with all Fs (all the bits are set). This process will reveal address lines that are stuck high.

Step 3. Beginning at STARTING ADDRESS and proceeding upward to STOP ADDRESS, each memory location is checked for bits that did not set, and then cleared to 0. This process will reveal address lines that are stuck low.

6.6.2 Command Input

M20Diag>MT E <CR>

6.6.3 Response/Messages

After the command is entered, the display should appear as follows:

E March Addr. Test..... Running --->

If an error is encountered, then the memory location and other related information will be displayed (see section 6.13).

E March Addr. Test..... Running --->

(error-related information) ..... FAILED

If no errors are encountered, then the display will appear as follows:

E March Addr. Test..... Running ---> PASSED

6.7 Walk A Bit Test MT F

6.7.1 Description

This command performs a "walking bit" test from STARTING ADDRESS to STOP ADDRESS.

The walking bit test has been implemented in the following manner:

```
For longwords from STARTING ADDRESS to STOP ADDRESS
    Test value = 1 (only lowest bit set in 32-bit value)
    For 32 bit positions
        Write test value to memory
        Read it back
        Verify that the value written equals the one read
        Report any errors
        Shift the 32-bit value to move the bit up one position
    Repeat
Repeat
```

6.7.2 Command Input

```
M20Diag>MT F <CR>
```

6.7.3 Response/Messages

After the command is entered, the display should appear as follows:

```
F Walk a Bit Test.............. Running --->
```

If an error is encountered, then the memory location and other related information will be displayed (see section 6.13).

```
F Walk a Bit Test.............. Running --->

(error-related information) ..... FAILED
```

If no errors are encountered, then the display will appear as follows:

```
F Walk a Bit Test.............. Running ---> PASSED
```

6.8 Refresh Test MT G

6.8.1 Description

This command performs a refresh test from STARTING ADDRESS to STOP ADDRESS.

The refresh test has been implemented in the following manner:

Step 1. For each memory location:
Write out Value $FC84B730.
Verify that the location contains $FC84B730.
Proceed to next memory location.

Step 2. Delay for 500 milliseconds (1/2 second).

Step 3. For each memory location:
Verify that the location contains$FC84B730.
Write out the complement of $FC84B730 ($037B48CF).
Verify that the location contains $037B48CF.
Proceed to next memory location.

Step 4. Delay for 500 milliseconds.

Step 6. For each memory location:
Verify that the location contains $037B48CF.
Write out value $FC84B730.
Verify that the location contains $FC84B730.
Proceed to next memory location.

6.8.2 Command Input

M20Diag>MT G

6.8.3 Response/Messages

After the command is entered, the display should appear as follows:

G Refresh Test..... Running --->

If an error is encountered, then the memory location and other related information will be displayed (see section 6.13).

G Refresh Test..... Running --->

(error-related information)

If no errors are encountered, then the display will appear as follows:

G Refresh Test..... Running ---> PASSED

6.9 Random Byte Test MT H

6.9.1 Description

This command performs a random byte test from STARTING ADDRESS to STOP ADDRESS.

The random byte test has been implemented in the following manner:

Step 1. A register is loaded with the value $ECA86420.

Step 2. For each memory location:
Copy the content of the register to the memory location, one byte at a time.
Add $02468ACE to the contents of the register.
Proceed to the next memory location.

Step 3. Reload $ECA86420 into the register.

Step 4. For each memory location:
Compare the contents of the memory to the register to verify that the contents are good, one byte at a time.
Add $02468ACE to the contents of the register.
Proceed to the next memory location.

6.9.2 Command Input

M20Diag>MT H

6.9.3 Response/Messages

After the command is entered, the display should appear as follows:

H Random Byte Test..... Running --->

If an error occurs, then the memory location and other information will be displayed (see section 6.13).

H Random Byte Test..... Running --->

(error-related information) ..... FAILED

If no errors occur, then the display will appear as follows:

H Random Byte Test..... Running ---> PASSED

6.10 Program Test MT I

6.10.1 Description

This command moves a program segment into RAM and executes it. The implementation of this is as follows:

Step 1. The program is moved into the RAM, repeating it as many times as necessary to fill the available RAM (i.e., from STARTING ADDRESS to STOP ADDRESS-8).

Note: Only complete copies of the program are moved. The space remaining from the last program segment copied into the RAM to STOP ADDRESS-8 is filled with NOP instructions. Attempting to run this test without sufficient memory (around 400 bytes) for at least one complete program segment to be copied will cause an error message to be printed out: "INSUFFICIENT MEMORY".

Step 2. The last location, STOP ADDRESS receives an RTS instruction.

Step 3. Finally, the test performs a JSR to location STARTING ADDRESS.

Step 4. The program itself performs a wide variety of operations, with the results frequently being checked and a count of the errors maintained. Errant locations are reported in the same fashion as any memory test failure (see section 6.13).

6.10.2 Command Input

```
M20Diag>MT I
```

6.10.3 Response/Messages

After the command is entered, the display should appear as follows:

```
I Program Test..................... Running --->
```

If the operator has not allowed enough memory for at least one program segment to be copied into the target RAM, then the following error message will be printed. To avoid this, make sure that the Stop Address is at least 388 bytes ($00000184) greater that the Start Address.

```
I Program Test..................... Running --->
Insufficient Memory
PASSED
```

If the program (in RAM) detects any errors, then the location of the error and other information will be displayed (see section 6.13).

```
I Program Test..................... Running --->
(error-related information) ..... FAILED
```

If no errors occur, then the display will appear as follows:

```
I Program Test..................... Running ---> PASSED
```

6.11 TAS Test MT J

6.11.1 Description

This command performs a Test And Set (TAS) test from STARTING ADDRESS to STOP ADDRESS.

The test is implemented as follows:

For each memory location:
- Clear the memory location to 0.
- Test And Set the location (should set upper bit only).
- Verify that the location now contains $80.
- Proceed to next location (next byte).

6.11.2 Command Input

```
M20Diag>MT J
```

6.11.3 Response/Messages

After the command is entered, the display should appear as follows:

```
J TAS Test.....................Running --->
```

If an error occurs, then the memory location and other information are displayed (see section 6.13).

```
J TAS Test.....................Running --->
(error-related information) ..... FAILED
```

If no errors occur, then the display will appear as follows:

```
J TAS Test.....................Running ---> PASSED
```

6.12 Test 0000-1FFF MT K

6.12.1 Description

This command performs the full suite of RAM tests on memory from $0000 through $1FFF. For each test D through J, the working contents of low memory (monitor stack, static variables, and vector table) are saved at $4000-$5FFF, and the test command is invoked with bounds of $0000 and $1FFF. The working contents are restored when each test is completed. If an error is detected, the test values in low memory are saved at $6000-$7FFF, the working contents are restored, the error is reported, the working contents are saved again, the test values are restored, and the test continues. Anything stored at $4000-$7FFF will be lost if MT K is run. Each test is logged to the console as it is begun.

6.12.2 Command Input

```
M20Diag>MT K
```

6.12.3 Response/Messages

After the command is entered, the display should appear as follows:

```
K Test 0000-1FFF.................Running --->

 - Random Inversion Test
 - March Address Test
 - Walk A Bit Test
 - Refresh Test
 - Random Byte Test
 - Program Test
 - TAS Test
```

If an error occurs, then the memory location and other information are displayed (see section 6.13). Errors detected by the invoked tests are accumulated; if any of the invoked tests detects an error, MT K fails.

```
K Test 0000-1FFF.................Running --->

 - Random Inversion Test
 - March Address Test
 - Walk A Bit Test
(error-related information)
 - Refresh Test
 - Random Byte Test
 - Program Test
 - TAS Test
 ..... FAILED
```

If no errors occur, then the display will appear as follows:

```
K Test 0000-1FFF.................Running --->

 - Random Inversion Test
 - March Address Test
 - Walk A Bit Test
 - Refresh Test
 - Random Byte Test
 - Program Test
 - TAS Test
PASSED
```

6.13 Partial Longword Writes Test MT L

6.13.1 Description

This command tests for a control logic failure during the write of a partial longword to RAM. The 68020 always drives all 32 data lines of the bus during a write cycle, but if the write is anything other than an aligned longword write, one or more of these bytes should be disregarded by the RAM array. The 68020 can signal to the RAM array which bytes are actually being written. See section 5.1.4, "Address, Size, and Data Bus Relationships" in the 68020 User's Manual for a complete explanation. If the RAM array control logic fails to interpret these signals correctly, data may not be written correctly to RAM. This test performs all possible partial longword writes to the memory array, and checks for errors. Since each half of the memory array has its own control logic, the test is repeated separately for both halves.

The test is performed by loading a register with the pattern $87654321, and performing nine writes to a target longword: byte writes to byte 0, byte 1, byte 2, and byte 3, word writes to bytes 0 and 1, bytes 1 and 2, and bytes 2 and 3, and three-byte writes to bytes 0, 1, and 2, and bytes 1, 2, and 3. Before each write, the target longword is zeroed, and after the write, the contents of the longword are compared to a copy of the correct result. If the contents don't match up, the error is reported as shown below.

6.13.2 Command Input

```
M20Diag>MT L
```

6.13.3 Response/Messages

After the command is entered, the display should appear as follows:

```
L Partial Longword Writes Test.. Running --->
```

If no errors occur, then the display will appear as follows:

```
L Partial Longword Writes Test.. Running ---> PASSED
```

If an error is encountered, then the error will be reported as follows:

```
L Partial Longword Writes Test.. Running --->
Error on <size> write to <location>
Expected pattern: 65432100
Actual   pattern: 65432187
```

If the error caused data to be written to RAM when it should have been ignored, this will show up as one or more zero bytes over non-zero bytes. If the error caused data to be ignored when it should have been written to RAM, this will show up as one or more non-zero bytes above zero bytes.

A fault of this type will usually render the GMX Micro-20 unusable till repaired. But if one's program avoids odd-address writes, or stores data only as aligned long words (as many 68020 programs do), the fault may be masked almost completely, and cause very subtle and confusing errors.

6.14 Description of Memory Error Display Format

This section is included to describe the format used to display errors during memory tests D - K.

The error reporting code is designed to conform to two rules:

1) The first time an error occurs, headings are printed out before the values are printed

2) After 20 memory errors, the printing of error messages ceases for the remainder of the test.

The figure below is included as an example of the display format.

```
FC  TEST ADDR   10987654321098765432109876543210   EXPECTED READ
 5  00010000    ----------------------X--------   00000100 00000000
 5  00010004    ------------------X-------X----   FFFFEFFF FFFFFEFF
```

Each line displayed consists of five items: function code, test address, graphic bit report, expected data, and read data. The test address, expected data, and read data are displayed in hexadecimal. The graphic bit report shows a letter 'X' at each errant bit position and a dash ('-') at each "good" bit position.

The heading used for the graphic bit report is intended to make the bit position easy to determine. Each numeral in the heading is the one's digit of the bit position. For example, the leftmost "bad" bit at test address $10004 has the numeral 2 over it. Since this is the second "2" from the right, the bit position is read "12" (base 10).

7.0 SIO TESTS

7.1 General Description

This set of tests exercises the GMX Micro-20's serial I/O hardware (MC68681 DUARTs). Both MC68681s (four ports) on the GMX Micro-20 board are tested automatically. Four or eight additional MC68681s on one or two SBC-8S serial I/O expansion boards can be tested as well.

TABLE T-2. SERIAL I/O DIAGNOSTIC TESTS

7.2 Hardware Requirements

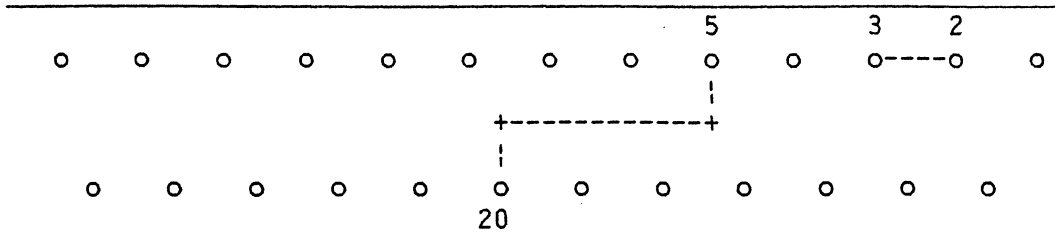
Tests SI B, SI D, SI E, SI F, SI H, and SI I use the internal loopback function of the MC68681. No external hardware is required for these tests.

Tests SI C and SI G require an external loopback circuit for each port which connects the Tx output to the Rx input and the DTR output to the CTS input. This circuit can be provided by plugging a "loopback connector" into the corresponding D-connector on the GMX RS-232 Adapter Board. A male DB-25 connector must be used with the SAB-25 Adapter board. A female DB-9 connector must be used with the SAB-9D and SAB-NT Adapter boards. The correct wiring pattern for each connector type is shown on the next page.

The fourth connector on an SAB-25 or SAB-9D adapter board has two additional handshake lines and jumpers for configuring the connector pinout. This supports applications such as modem control. The jumpers must be positioned correctly for the loopback connector to function properly.

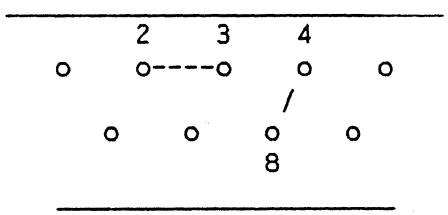
The fourth connector on an SAB-NT board is wired to support the RS-485 high speed interface and network operations. Its transmit and receive lines can be "looped back" externally through the network support circuitry, but its handshake lines cannot be "looped back" at all. Therefore no loopback connector is needed or usable with this connector, and the SI G test will skip the corresponding DUART channel if an SAB-NT board is installed.

Wiring of DB-25 male loopback connector



Connect			
From		To	
2	Rx data	3	Tx data
5	DTR	20	CTS

Wiring of DB-9 female loopback connector



Connect			
From		To	
2	Rx data	3	Tx data
8	DTR	4	CTS

7.3 Select DUARTs for testing SI A

7.3.1 Description

This command allows the user to select DUARTs for testing and determine which DUARTs are installed in the system. Each possible DUART address in the range <low DUART> through <high DUART> is accessed by the MPU. If no bus error occurs that DUART is flagged as "installed" and selected for testing. If only one DUART number is specified, only that DUART is tested and flagged. If no range is given, the default range is 0 through 1. The user is then informed which DUARTs are selected. The permitted DUART numbers are from 0 to 17. 0 and 1 are the on-board DUARTs. 2 through 17 are located on expansion boards. When a specific DUART test is selected it will be performed for all DUARTs which are selected.

SI A may be used to test the address decoding and bus handshake logic of the I/O expansion port and boards, or of DUART #1 (if DUART #0 malfunctions console I/O is impossible).

When an SAB-NT Adapter Board is used, one DUART will have the RS-485/network interface on its channel A, which does not allow external loopback of the handshake lines. The SI A command tests for the network interface, and any DUART which is connected to the RS-485 connector of an SAB-NT board will be flagged with a "*" after its number in SI A's report line.

7.3.2 Command Input

```
M20Diag>SI A [low DUART [high DUART]]
```

7.3.3 Response/Messages

After the command line is entered a response should appear as follows:

```
M20Diag>SI A 1 2 <CR>                          (no expansion boards installed)
These DUARTs are selected: 1
M20Diag>

M20Diag>SI A <CR>                              (default to 0 through 1)
These DUARTs are selected: 0 1
M20Diag>

M20Diag>SI A 0 11 <CR>                         (one expansion board
These DUARTs are selected: 0 1 2 3* 4 5 6 7 8 9 installed; RS-485 interface
M20Diag>                                       on DUART #3 channel A)

M20Diag>SI A 1 D <CR>                          (two expansion boards
These DUARTs are selected: 1 6 7 8 9 A B C D   installed, decoding failure
M20Diag>                                       on first board)

M20Diag>SI A 2 <CR>                            (expansion board installed)
These DUARTs are selected: 2
M20Diag>

M20Diag>SI A 1 <CR>                            (DUART #1 not responding)
These DUARTs are selected:
M20Diag>
```

7.4 Internal loopback function SI B

7.4.1 Description

This test uses the 68681's internal loopback functiom to transmit a 256-byte buffer of pseudo-random data from the send side to the receive side of each channel of each DUART selected for testing. The data rate is 19200 baud.

7.4.2 Command Input

```
M20Diag>SI B
```

7.4.3 Response/Messages

After the command is entered, the display should appear as follows:

```
B Internal loopback function.... Running --->
```

If an error is encountered, then the related information will be displayed (see section 7.12). The display will then appear as follows:

```
B Internal loopback function.... Running --->
Now testing DUART #1
(error-related information)
..... FAILED
```

If no errors occur, then the display will appear as follows:

```
B Internal loopback function.... Running --->
Now testing DUART #1
PASSED
```

7.5 External loopback connector SI C

7.5.1 Description

This test transmits a 256-byte buffer of pseudo-random data from the send side to the receive side of each channel through an external loopback circuit (see section 7.2, Hardware Requirements). Both channels of all selected DUARTs are tested except channel A of DUART #1, which is used by the console. This test does NOT exercise the external loopback of the handshake lines, which must be checked with the SI G command.

When an SAB-NT adapter board is used, this test will exercise the high-speed external clock input from the nework interface to the the DUART.

7.5.2 Command Input

```
M20Diag>SI C
```

7.5.3 Response/Messages

After the command is entered, the display should appear as follows:

```
C External loopback connector... Running --->
```

If an error is encountered, then the related information will be displayed (see section 7.12). The display will then appear as follows:

```
C External loopback connector... Running --->
Now testing DUART #1
(error-related information)
..... FAILED
```

If no errors occur, then the display will appear as follows:

```
C External loopback connector... Running --->
Now testing DUART #1
PASSED
```

7.6 Baud rates SI D

7.6.1 Description

This test exercises the baud rate generation functions of each DUART by sending a 256 byte buffer through the internal loopback circuit on both the A and B channels. Baud rates from 75 baud to 38400 baud are tested, and all selected DUARTs are tested.

7.6.2 Command Input

```
M20Diag>SI D
```

7.6.3 Response/Messages

After the command is entered, the display should appear as follows:

```
D Baud rates.....................Running --->
```

If an error is encountered, then the baud rate will be displayed along with other related information (see section 7.12). The display will then appear as follows:

```
D Baud rates.....................Running --->
Now testing DUART #1
(error-related information)
Baud rate = 19200
..... FAILED
```

If no errors occur, then the display will appear as follows:

```
D Baud rates.....................Running --->
Now testing DUART #1
PASSED
```

7.7 Parity modes SI E

7.7.1 Description

This test exercises the four parity modes (odd, even, space, mark, and none) for both channels of all the selected DUARTs by sending a 256 byte buffer through the internal loopback circuit in each mode on both channels of each DUART.

7.7.2 Command Input

```
M20Diag>SI E
```

7.7.3 Response/Messages

After the command is entered, the display should appear as follows:

```
E Parity modes..................... Running --->
```

If an error is encountered, then the parity mode and other error related information will be displayed (see section 7.12). The display will then appear as follows:

```
E Parity modes..................... Running --->
Now testing DUART #1
(error-related information)
Even parity
..... FAILED
```

If no errors occur, then the display will appear as follows:

```
E Parity modes..................... Running --->
Now testing DUART #1
PASSED
```

7.8 Character lengths SI F

7.8.1 Description

This test exercises the different character lengths of the MC68681 (5, 6, 7, or 8 bits/character) by sending a 256 byte buffer through the internal loopback circuit of both channels each selected DUART.

7.8.2 Command Input

```
M20Diag>SI F
```

7.8.3 Response/Messages

After the command is entered, the display should appear as follows:

```
F Character lengths..... Running --->
```

If an error is encountered, then the character length and other error related information will be displayed (see section 7.12). The display will then appear as follows:

```
F Character lengths..... Running --->
Now testing DUART #1
(error-related information)
5 bits per character
..... FAILED
```

If no errors occur, then the display will appear as follows:

```
F Character lengths..... Running --->
Now testing DUART #1
PASSED
```

7.9 Handshake lines SI G

7.9.1 Description

This test exercises the handshake functions of each DUART by toggling the handshake output and examining the handshake input of both channels port. An external loopback connector must be installed for this test to work. Both channels of all selected DUARTs are tested except channel A of DUART #0, which is used by the console, and any other DUART's channel A which is attached to a RS-485 interface. External loopback connectors must be installed for this test to work correctly.

7.9.2 Command Input

```
M20Diag>SI G
```

7.9.3 Response/Messages

After the command is entered, the display should appear as follows:

```
G Handshake lines ..... Running --->
```

If the handshake lines do not operate correctly, then the channel location and error type will be reported as follows:

```
G Handshake lines ..... Running --->
Now testing DUART #1
Handshake error on DUART #1 channel A
Handshake line stuck low
..... FAILED
```

If no errors occur, then the display will appear as follows:

```
G Handshake lines ..... Running --->
Now testing DUART #1
PASSED
```

7.10 BREAK detect SI H

7.10.1 Description

This test exercises the BREAK detect and BREAK transmit functions of the MC68681 by sending a BREAK on the internal loopback circuit of both channels of each selected DUART.

7.10.2 Command Input

```
M20Diag>SI H
```

7.10.3 Response/Messages

After the command is entered, the display should appear as follows:

```
H BREAK detect..................... Running --->
```

If BREAK is not correctly detected, a message reporting this will be printed on the screen, and the display will appear as follows

```
H BREAK detect..................... Running --->
Now testing DUART #1
No DELTA BREAK detect on DUART #1 channel A
..... FAILED
```

or

```
H BREAK detect..................... Running --->
Now testing DUART #1
No valid BREAK flag in channel status on DUART #1 channel A
..... FAILED
```

If no errors occur, then the display will appear as follows:

```
H BREAK detect..................... Running --->
Now testing DUART #1
PASSED
```

7.11 Interrupt generation SI I

7.11.1 Description

This test exercises the interrupt output of the DUART by causing a TxRDY interrupt from both channels of each selected DUART using the internal loopback circuit.

7.11.2 Command Input

```
M20Diag>SI I
```

7.11.3 Response/Messages

After the command is entered, the display should appear as follows:

```
I Interrupt generation..... Running --->
```

If no interrupt is generated, the error will be reported and the display will then appear as follows:

```
I Interrupt generation..... Running --->
Now testing DUART #1
No interrupt on DUART #1 channel A
..... FAILED
```

If no errors occur, then the display will appear as follows:

```
I Interrupt generation..... Running --->
Now testing DUART #1
PASSED
```

7.12 Handshake toggle SI J

7.12.1 Description

This test performs a continuous toggle of both handshake lines of all selected DUARTs, except DUART#0 channel A (the console port). All the handshake lines are set high one at a time, starting from the lowest selected DUART. Then the lines are set back low in the same order. This process will continue until a BREAK is entered on the console. This command is useful to a technician who needs to examine the handshake lines changing state.

7.12.2 Command Input

```
M20Diag>SI J
```

7.12.3 Response/Messages

After the command is entered, the display should appear as follows:

```
J Handshake toggle..... Running --->
```

A standard "!!Break!!" message is displayed when the user terminates the command by entering a BREAK on the console.

7.13 SIO Error Reporting

When an error is detected by one of the SI tests, the error is reported to the user on the screen. In addition to reporting the type of error, the test will also display the number of the DUART which displayed the fault and which channel failed.

For tests which send a data buffer through an external or internal loopback circuit, additional error conditions are reported. These include timeout while waiting to transmit or receive, and parity, framing, or overrun errors reported by the MC68681. These error conditions are indicated separately, and the number of bytes previously received is reported.

Thus, if the 53rd byte in a buffer being sent to test the external loopback connector generates a parity error, the display would be as follows:

```
M20Diag>SI C
Now testing DUART #1
Receiver error on DUART #1 channel A at byte 053
Parity error
```

If a buffer transmission is completed without errors detected, then the received data buffer is compared against the sent data buffer. If a mismatch is found, the buffer position, byte sent, and byte received are displayed. A maximum of twenty data errors will be reported. If the 254th byte of the buffer was received with a wrong value, the display would be as follows:

```
Data error on DUART #1 channel A at byte 254
Position  Sent  Received
   FE      AB     A7
```

8.0 MC68020 (ON-CHIP) CACHE TESTS

8.1 General Description

This section details the diagnostics provided to test the MC68020 cache.

TABLE T-2. MC68020 CACHE DIAGNOSTIC TESTS

Monitor Command	Title	Section	Page
CA20 A	Basic Caching	8.2	D-36
CA20 B	Unlike Function Codes	8.3	D-37
CA20 C	Disable Test	8.4	D-38
CA20 D	Clear Test	8.5	D-39

The normal procedure for fixing a MC68020 cache error is to replace the MPU.

8.2 Basic Caching Test CA20 A

8.2.1 Description

This command tests the basic caching function of the MC68020 microprocessor. The test caches a program segment that resides in RAM, freezes the cache, changes the program segment in RAM, then re-runs the program segment. If the cache is functioning correctly, the cached instructions will be executed. Failure is detected if the MC68020 executes the instructions that reside in RAM; any cache misses will cause an error.

The process is first attempted in supervisor mode for both the initial pass through the program segment and the second pass. It is then repeated, using user mode for the initial pass and the second pass. A bit is included in each cache entry for distinguishing between supervisor and user mode. If this bit is stuck or inaccessible, the cache will miss during one of these two tests.

8.2.2 Command Input

```
M20Diag>CA20 A
```

8.2.3 Response/Messages

After the command is entered, the following line will be printed:

```
A Basic Caching ..................... Running --->
```

If there are any cache misses during the second pass through the program segment, then the test fails and the display will appear as follows.

```
A Basic Caching ..................... Running --->
2 CACHE MISSES!
CACHED IN SUPY MODE, RERAN IN SUPY MODE
..... FAILED
```

If there are no cache misses during the second pass, then the test passes.

```
A Basic Caching ..................... Running ---> PASSED
```

8.3 Unlike Function Codes Test CA20 B

8.3.1 Description

This command tests the ability of the on-chip cache to recognize function codes. Bit 2 of the function code is included in the tag for each entry. This provides a distinction between supervisor and user modes for the cached instructions. To test this mechanism, a program segment that resides in RAM is cached in supervisor mode. The cache is frozen, then the program segment in RAM is changed. When the program segment is executed a second time in user mode, there should be no cache hits due to the different function codes. Failure is detected if the MC68020 executes the cached instructions.

After the program segment has been cached in supervisor mode and rerun in user mode, the process is repeated, caching in user mode and re-running in supervisor mode. Again, the cache should miss during the second pass through the program segment.

8.3.2 Command Input

```
M20Diag>CA20 B
```

8.3.3 Response/Messages

After the command is entered, the following line will be printed:

```
B Unlike fn. Codes ..... Running --->
```

If there are any cache hits during the second pass through the program segment, then the test fails and the display will appear as follows.

```
B Unlike fn. Codes ..... Running --->
5 CACHE HITS!
CACHED IN SUPY MODE, RERAN IN USER MODE
..... FAILED
```

If there are no cache misses during the second pass, then the test passes.

```
B Unlike fn. Codes ..... Running ---> PASSED
```

8.4 Disable Test CA20 C

8.4.1 Description

In the MC68020 cache control register ("CACR") a control bit is provided to enable the cache. When this bit is clear, the cache should never hit, regardless of whether the address and function codes match a tag. To test this mechanism, a program segment is cached from RAM. The cache is frozen to pre serve its contents, then the enable bit is cleared. The program segment in RAM is then changed and rerun. There should be no cache hits with the enable bit clear. Failure is declared if the cache does hit.

8.4.2 Command Input

```
M20Diag>CA20 C
```

8.4.3 Response/Messages

After the command is entered, the following line will be printed:

```
C Disable Test ..................... Running --->
```

If there are any cache hits during the second pass through the program segment, then the test fails and the display will appear as follows.

```
C Disable Test ..................... Running --->
1 CACHE HIT!
CACHED IN SUPY MODE, RERAN IN SUPY MODE
..... FAILED
```

If there are no cache misses during the second pass, then the test passes.

```
C Disable Test ..................... Running ---> PASSED
```

8.5 Clear Test CA20 D

8.5.1 Description

A control bit is included in the MC68020 Cache Control register ("CACR") to clear the cache. Writing a one to this bit invalidates every entry in the on-chip cache. To test this function, a program segment in RAM is cached and then frozen there to preserve it long enough to assert the cache clear control bit. The program segment in RAM is then modified and rerun with the cache enabled. If the cache hits, the clear is incomplete and failure is declared.

8.5.2 Command Input

```
M20Diag>CA20 D
```

8.5.3 Response/Messages

After the command is entered, the following line will be printed:

```
D Clear Test ..................... Running --->
```

If there are any cache hits during the second pass through the program segment, then the test fails and the display will appear as follows.

```
D Clear Test ..................... Running --->
58 CACHE HITS!
CACHED IN SUPY MODE, RERAN IN SUPY MODE
..... FAILED
```

If there are no cache misses during the second pass, then the test passes.

```
D Clear Test ..................... Running ---> PASSED
```

9.0 GMX MICRO-20 MISCELLANEOUS HARDWARE TESTS

9.1 General Description

This section details the diagnostics provided to test various hardware functions of the GMX Micro-20.

TABLE T-3. GMX MICRO-20 MISCELLANEOUS HARDWARE TESTS

Monitor Command	Title	Section	Page
MH A	FPC instructions	9.2	D-41
MH B	FPC control functions	9.2	D-42
MH C	Tick generator	9.3	D-44
MH D	Interrupt generation	9.4	D-45

9.2 FPC Instructions MH A

9.2.1 Description

This test checks the functioning of the MC68881 floating-point coprocessor and its interface circuitry. Bit 6 of the SASI status register at $FFFF800E is a 1 if the coprocessor is installed. If this bit is a 0, then the test terminates with a message and no error is generated.

If the coprocessor is installed, then each of the arithmetic instructions of the MC68881 floating-point coprocessor are executed, using preset operand values and checking the results against known correct values. If a result does not match, then the erring instruction is reported.

The normal procedure for fixing a coprocessor error is to replace the coprocessor.

9.2.2 Command Input

```
M20Diag>MH A
```

9.2.3 Response/Messages

After the command is entered, the following line will be printed:

```
A FPC instructions ..... Running --->
```

If no FPC is installed the display will look like this:

```
A FPC instructions ..... Running --->
No coprocessor installed
PASSED
```

If a coprocessor instruction produces a wrong result, the display will look like this:

```
A FPC instructions ..... Running --->
Floating point error in FSIN    instruction
..... FAILED
```

If the coprocessor is installed, but does not execute instructions, the display will look like this:

```
A FPC instructions ..... Running --->
F-line exception - Coprocessor not responding
..... FAILED
```

If there are no wrong results generated, then the test passes.

```
A FPC instructions ..... Running ---> PASSED
```

9.3 FPC control functions MH B

9.3.1 Description

This test exercises the control functions of the MC68881 floating-point coprocessor. Like the MH A test, it exits without generating an error if no FPC is installed. If the coprocessor is installed, a series of tests is performed which exercises all of the format conversion functions, FPC exception generation, FPC save and restore, and the floating-point condition code bits. An error message is printed if any of these functions does not work correctly.

The normal procedure for fixing a coprocessor error is to replace the coprocessor.

9.3.2 Command Input

```
M20Diag>MH B
```

9.3.3 Response/Messages

After the command is entered, the following line will be printed:

```
B FPC control functions ..... Running --->
```

If no FPC is installed the display will look like this:

```
B FPC control functions ..... Running --->
No coprocessor installed
PASSED
```

If a coprocessor control function does not work correctly, the display will look like this:

```
B FPC control functions ..... Running --->
Control failure in floating-point coprocessor
..... FAILED
```

If all the control functions work correctly, then the test passes and the display will look like this:

```
B FPC control functions ..... Running ---> PASSED
```

9.4 Tick generator MH C

9.4.1 Description

The GMX Micro-20 has a tick generator which produces level 6 Autovector interrupts at regular intervals. (See the GMX Micro-20 Hardware Manual for more details.) This test checks the operation of the tick generator by turning it on and counting the ticks generated in 20 seconds. The 20 second interval is measured using the BUSY output of the time of day clock. The tick generator is factory jumpered for 10 millisecond tick intervals, so 2000 ticks should be generated in 20 seconds. There is a small amount of error possible in the length of 20 second interval, so the count may be one tick more or less than 2000. Therefore if the count is 1999, 2000, or 2001 ticks the test is passed.

If the count is not in this range, then the count is reported and an error is recorded. If the user has jumpered the board for a different tick rate, this test can be used to check the result. It is also possible that the time-of-day clock is malfunctioning.

The tick generator uses the 68230 PI/T to count ticks directly, so the system can compensate for ticks which occur while the level 6 interrupt is masked. Also, a rollover flag is provided for cases where level 6 is masked so long that the counter overflows. These functions are also exercised in this test.

9.4.2 Command Input

```
M20Diag>MH C
```

9.4.3 Response/Messages

After the command is entered, the following line will be printed:

```
C Tick generator ..................... Running --->
```

If the count of tick interrupts is not 1999, 2000, or 2001, an error message will be displayed:

```
C Tick generator ..................... Running --->
1234 ticks generated per second
..... FAILED
```

If the tick count in the PI/T is not correct, an error message will be displayed:

```
C Tick generator ..................... Running --->
Hard tick counter not counting correctly
..... FAILED
```

If the overflow bit in the PI/T is not set when it should be, an error message will be displayed:

```
C Tick generator ..................... Running --->
No rollover flag on hard tick counter
..... FAILED
```

Finally, if the time of day clock never sets its BUSY output the test will be aborted with this error message:

```
C Tick generator ..................... Running --->
Test aborted - time of day clock not working
..... FAILED
```

9.5 Interrrupt generation MH D

9.5.1 Description

There are seven on-board devices in the GMX Micro-20 which generate Autovector interrupts. This test exercises the interrupt generation function for three of these devices which can be tested without any external hardware, and are not otherwise tested in the diagnostics: the Floppy Disk Controller (level 5), the 68230 Timer section (level 4), and the 68681 DUARTs (level 3). Only one DUART is tested.

The test program causes each of these devices to generate its Autovector interrupt, and then checks to see that the interrupt was generated at the correct level, and that no other interrupts were generated.

9.5.2 Command Input

```
M20Diag>MH D
```

9.5.3 Response/Messages

After the command is entered, the following line will be printed:

```
D Interrupt generation..... Running --->
```

If the expected interrupt was not generated, an error message will be printed:

```
D Interrupt generation..... Running --->
Did not receive level 5 interrupt
..... FAILED
```

If the interrupt generated was the wrong level, an error message will be printed:

```
D Interrupt generation..... Running --->
Wrong interrupt - expected 5 received 4
..... FAILED
```

If the Spurious Interrupt Exception was generated instead of an Autovector interrupt, an error message will be printed:

```
D Interrupt generation..... Running --->
Spurious interrupt instead of level 5
..... FAILED
```

10.0 PARALLEL PORT TESTS

10.1 General Description

This section describes the tests functions available for exercising the GMX Micro-20's parallel I/O port. Unlike the serial I/O tests, these tests do not detect or report errors. They are used to send test patterns to the parallel port which can be checked by examining printout or by testing with an oscilloscope.

TABLE T-4. PARALLEL PORT DIAGNOSTIC TESTS

Monitor Command	Title	Section	Page
PP A	Print test pattern on parallel port	10.2	D-47
PP B	Send test bit pattern to parallel port	10.3	D-48
PP C	Send test bit pattern to PI/T port	10.4	D-49

10.2 Print test pattern on parallel port PP A

10.2.1 Description

This command sends a continuous stream of ASCII characters to the parallel output port. If the port is configured for "Centronics" parallel output operation, and a printer is connected, then a "barberpole" pattern will be printed by the printer. This pattern will be 80 columns wide, and includes the characters ' ' (space) through DEL. Printing will continue until the user enters a BREAK on the console.

10.2.2 Command Input

```
M20Diag>PP A
```

10.2.3 Response/Messages

After the command is entered, the display will look like this:

```
A Print test pattern on parallel port Running --->
```

A standard "!!Break!!" message will be displayed when the user types a BREAK to exit the command.

10.3 Send test bit pattern to parallel port PP B

10.3.1 Description

This command puts a rotating bit pattern on the parallel output port, consisting of seven 0s and a 1. The 1 starts in bit position 0, and in each successive byte is shifted one position higher, wrapping around from position 7 to position 0. Meanwhile, a technician with an oscilloscope can examine the outputs individually to check for open, shorted, or coupled lines. Output of the test pattern will continue until the user types a BREAK on the console.

10.3.2 Command Input

```
M20Diag>PP B
```

10.3.3 Response/Messages

After the command is entered, the display will look like this:

```
B Send test bit pattern to parallel port Running --->
```

A standard "!!Break!!" message will be displayed when the user types a BREAK to exit the command.

10.4 Send test bit pattern to PI/T port PP C

10.4.1 Description

This command sends the same test pattern as the PP C command to the parallel output port, but automatically stops after 10-20 seconds (depending on processor speed). Typing a BREAK does not interrupt this command. It is coded as part of the self-test sequence to give the parallel outputs some exercise while a GMX Micro-20 is burning in.

10.4.2 Command Input

```
M20Diag>PP C
```

10.4.3 Response/Messages

After the command is entered, the display will look like this:

```
C Send test bit pattern to PI/T port Running --->
```

When the test terminates, a "PASSED" message will always be displayed.

11.0 FLOPPY DISK CONTROLLER TESTS

11.1 General Description

This section describes the tests functions available for exercising the GMX Micro-20's WD1772 Floppy Disk Controller. For these tests to be useful, a floppy disk drive should be connected to the GMX Micro-20's floppy disk interface. The tests support single or double sided operation, single or double density data recording, 40 or 80 track drive configuration, and one or two drives. However, the WD1772 only supports 5 1/4" drives or equivalents.

The FDC commands use certain areas of RAM as buffers for read and write operations. The read buffer is at $20000, and the write buffer is at $10000. Up to 7 Kbytes of RAM at each of these addresses may be overwritten during FDC diagnostics.

Some of the FDC diagnostics execute WD1772 commands. Whenever a WD1772 command is executed, the resulting values of the WD1772 Status register and the GMX Micro-20 Control/Status Register (CTSR) are displayed in binary. See section 11.13 for further explanation of the various bits in these registers. For a more complete explanation of the concepts and hardware involved the user should obtain the data sheet for the WD1772.

TABLE T-5. FLOPPY DISK CONTROLLER DIAGNOSTIC TESTS

Monitor	Command	Title	Section	Page
FD	A	Set parameters	11.2	D-52
FD	B	Drive select	11.3	D-54
FD	C	Side select	11.4	D-55
FD	D	Restore	11.5	D-56
FD	E	Seek	11.6	D-57
FD	F	Format track	11.7	D-58
FD	G	Read sector	11.8	D-59
FD	H	Write sector	11.9	D-60
FD	I	Copy read buffer to write buffer	11.10	D-61
FD	J	Compare read and write buffers	11.11	D-62
FD	K	Fill write buffer with data	11.12	D-63

11.1.2 Hardware requirements

To use these tests, the system must have a standard 5 1/4" floppy disk drive or equivalent device connected to the floppy disk interface connector. Single or double density, single or double sided, and 40 or 80 track operations are all supported. Head stepping intervals from 2 to 12 milliseconds may be selected. A drive with a minimum step interval of more than 12 milliseconds will not work properly with these commands.

11.1.1 Floppy disk format

The user can format a track on a floppy with the FD F command. When this is done, a track format block is created in memory at $10000, and then written to the disk with a WD1772 Write Track command. The track format is standard IBM as described in the manufacturer's data sheet for the WD1772. Sectors on a track are numbered starting from 1. No interleave factor is used: the sectors are numbered in their physical order on the disk. The side number fields of the sector address blocks on the disk are filled, but the sector numbers start at 1 on both sides of the disk. The number of sectors created is the maximum allowed for the current data density and sector size: this number can be found in the following table.

Sector size	128	256	512	1024
Single density sectors per track	28	16	9	5
Double density sectors per track	16	9	5	2

11.2 Set parameters FD A

This command displays the parameters for floppy disk operations, and allows the user to change these parameters. It must be run before any other floppy disk operations are performed. All the parameters are entered as single characters. If the character typed is not a valid entry the prompt will be repeated. Example:

```
M20Diag>FD A
Drive     = 0 -------------------------- 0 or 1? 0
Density   = D ------------ S(ingle) or D(ouble)? D
Side      = 0 -------------------------- 0 or 1? 0
DRQ Disabled ----------- E(nable) or D(isable)? D
Step rate= 6 ms - 0 (6ms)  1 (12ms)  2 (2ms)  3 (3ms)? 0
No verify ------------ V(erify) or N(o verify)? N
Write precomp on --- P(recomp) or N(o precomp)? P
No settle delay -------- D(elay) or N(o delay)? N
# of tracks= 80 ----------------- 4(0) or 8(0)? 8
Sector size= 0128 - (0=128 1=256 2=512 3=1024)? 1
M20Diag>
```

The effect of each parameter is as follows:

```
Drive     = 0 -------------------------- 0 or 1?
```

This parameter sets one of two bits in the mask written to the CTSR when a WD1772 command is performed. It has no effect on the operation of the FD B command.

```
Density   = D ------------ S(ingle) or D(ouble)?
```

This parameter sets or clears a bit in the CTSR which determines the data clock rate of the WD1772 and the density of data recording on the disk.

```
Side      = 0 -------------------------- 0 or 1
```

This parameter sets or clears a bit in the CTSR which selects the side of the drive to be read or written.

```
DRQ Disabled ----------- E(nable) or D(isable)?
```

This parameter sets or clears a bit in the CTSR which enables the DRQ (Data ReQuest output as the level 7 Autovector Interrupt. DRQ is generated during any WD1772 Read or Write commands when a data byte is to be read from or written to the WD1772's data register. If DRQ is enabled, the FD F, FD G, and FD H commands are performed using interrupts; if DRQ is disabled, register polling is done instead.

```
Step rate= 6 ms - 0 (6ms)  1 (12ms)  2 (2ms)  3 (3ms)?
```

This parameter sets up a two bit mask which is used in WD1772 Restore and Seek commands to control the stepping rate of the floppy disk drive. To determine the best value for this parameter, the manufacturer's description of the drive should be consulted. "ms" stands for "milliseconds".

No verify ---------- V(erify) or N(o verify)?

This parameter sets or clears a bit which is masked into WD1772 Seek and Restore command bytes. It controls whether the WD1772 verifies each track movement operation by reading an address off the destination track, reporting an error if it cannot read an address which matches the updated internal Track register. It has nothing to do with the verification of a newly formatted track in the FD F command.

Write precomp on --- P(recomp) or N(o precomp)?

This parameter sets or clears a bit which is masked into WD1772 Write and Write Track commands and tells the WD1772 whether or not to perform precompensation while writing to the disk. This bit is used if it is set and the track number is greater than half the maximum specified with the FD A command.

No settle delay -------- D(elay) or N(o delay)?

This parameter sets or clears a bit which is masked into WD1772 Read and Write Track commands and tells the WD1772 whether or not to perform Write parameters and tells the WD1772 whether or not to wait 30 milliseconds for the head to load on the disk and settle into place.

# of tracks= 80 ---------- 4(0) or 8(0)?

This parameter sets or clears a flag which is used to determine the limit of track numbers for the Seek parameter and for deciding whether Write Precomp should be used.

Sector size= 0128 - (0=128 1=256 2=512 3=1024)?
This parameter sets up a two bit mask which is used in track formatting as the sector size field, and by various parameters to determine the size of a sector data field.

FD B

This command prompts the user for a drive number, entered as a single character. When the drive number is enters, the select line for that drive only is made active, and the prompt is repeated. If a CR is typed instead of a 0 or 1, the command terminates with both drives deselected. This command is useful for a technician checking for shorted or open lines. Example:

```
M20Diag>FD B
0=select drive 0  1=select drive 1  CR=exit  1          (selects drive 1)
0=select drive 0  1=select drive 1  CR=exit  0          (selects drive 0)
0=select drive 0  1=select drive 1  CR=exit  <cr>       (deselects both)
M20Diag>
```

FD C

This command toggles the side select line to the floppy disk interface. It stops and exits when the user types one character on the console. This command is useful for a technician checking for shorted or open lines. Example:

```
M20Diag>FD C
Side select toggling - Hit any key to exit
M20Diag>
```

FD D

This command executes a WD1772 Restore command using the defaults established by the FD A command. Example:

```
M20Diag>FD D
D Restore..................... Running --->
WD1772 status = 10100110  GMX status = 00011110
PASSED
M20Diag>
```

If the FD A command has not been performed (so that no drive is selected) or the selected drive is not connected, or the drive fails to go ready in 10 seconds, the display will look like this:

```
M20Diag>FD D
D Restore..................... Running ---> No drive selected

WD1772 status = 00000000  GMX status = 00111110
FAILED
M20Diag>
```

11.6 Seek FD E

FD E [<track>]

This command executes a WD1772 Seek command using the defaults established by the FD A command. The target track for the seek may be entered on the command line in hexadecimal. Example:

```
M20Diag>FD E 10
E Seek.......................... Running --->
WD1772 status = 10100000  GMX status = 00011110
PASSED
M20Diag>
```

If the target track is not entered on the command line, the user will be prompted for it. Example:

```
M20Diag>FD E
E Seek.......................... Running ---> Track number (0-$4F)? 10

WD1772 status = 10100000  GMX status = 00011110
PASSED
M20Diag>
```

The user will be reprompted if the entered track number exceeds the limit of the drive as specified by the FD A command.

This command assumes that the current value in the WD1772 Track register is the actual position of the selected drive's head. If the user's system has two drives, when the user switches between them this register is not updated and errors may be generated.

FD F

This command sets up a track format block using the defaults set by the FD A command, then writes it to the disk with the WD1772 Write Track command. No track number is specified: the track at the current head position will be formatted, and the track ID number is taken from the WD1772 Track register. The number of sectors created is the maximum allowed for the given density and sector size, as described in section 11.1.1. The data field of each sector is filled with the worst-case data pattern for the current data density; that is, the data pattern most likely to result in errors when read back. After the track write is finished, the track is verified by reading all sectors with a multiple sector Read command. Example:

```
M20Diag>FD F
E Format track..................... Running --->
WD1772 status = 10000000  GMX status = 00011110
Track written - verifying...
PASSED
M20Diag>
```

The status registers are not displayed after the track verify unless it terminated abnormally. The "Record Not Found" will be normally be set, as this is how the WD1772 terminates a multiple sector read. If the command terminated from some other cause or the wrong number of sectors was read back the registers will be displayed. Example:

```
M20Diag>FD F
F Format track..................... Running --->
WD1772 status = 10000000  GMX status = 00011110
Track written - verifying...

WD1772 status = 10001000  GMX status = 00011110
M20Diag>
```

FD G [<sector>]

This command executes a WD1772 Read command using the defaults established by the FD A command. The target track is simply the current head position; the target sector for the read may be entered on the command line in hexadecimal. The address of the read buffer is reported along with the status, and the user can use the MD command to display the buffer's contents. Example:

```
M20Diag>FD G 10
G Read.......................... Running --->
WD1772 status = 10100000  GMX status = 00011110
Read buffer at $20000-$2007F
PASSED
M20Diag>
```

If the target sector is not entered on the command line, the user will be prompted for it. Example:

```
M20Diag>FD G
G Read.......................... Running ---> Sector number (0-$4F)? 10

WD1772 status = 10100000  GMX status = 00011110
PASSED
M20Diag>
```

The user will be reprompted if the entered sector number is zero or exceeds the number of sectors per track allowed for the data density and sector size specified by the FD A command.

11.9 Write FD H

FD H [<sector>]

This command executes a WD1772 Write command using the defaults established by the FD A command. The target track is simply the current head position; the target sector for the write may be entered on the command line in hexadecimal. The address of the write buffer is reported along with the status, and the user can use the BF, MM, MS, or FDK commands to set up whatever sort of test data is desired in the buffer. Example:

```
M20Diag>FD H 10
H Write..... Running --->
WD1772 status = 10100000  GMX status = 00011110
Write buffer at $10000-$1007F
PASSED
M20Diag>
```

If the target sector is not entered on the command line, the user will be prompted for it. Example:

```
M20Diag>FD H
H Write..... Running ---> Sector number (0-$4F)? 10

WD1772 status = 10100000  GMX status = 00011110
PASSED
M20Diag>
```

The user will be reprompted if the entered sector number is zero or exceeds the number of sectors per track allowed for the data density and sector size specified by the FD A command.

11.10 Copy read buffer to write buffer FD I

FD I

This command provides a convenient way of moving data which has been read from a disk into position to be written back to the disk. It copies data from $20000 to $10000, and moves as many bytes as the sector length specified by the FD A command. No errors are possible.

11.11 Compare read and write buffers FD J

FD J

This command performs a byte by byte comparison of the contents of the read and write buffers. As many bytes are compared as the sector length specified by the FD A command. Example:

```
M20Diag>FD J
J Compare read and write buffers Running ---> PASSED
```

If any bytes do not match the differences are reported and an error is recorded. Example:

```
M20Diag>FD J
J Compare read and write buffers Running --->
Data error at byte 1D - data written = FF - data read = DB
Data error at byte 1E - data written = FF - data read = DB..... FAILED
M20Diag>
```

11.12 Fill write buffer with data FD K

FD K [<data>]

This command fills the write buffer with either a specified longword data pattern or the worst-case data pattern for the current data density. The address of the write buffer is reported. Example:

```
M20Diag>FD K 12349999
Write buffer at $10000-$1007F
M20Diag>MD 10000
00010000  1234 9999 1234 9999  1234 9999 1234 9999     .4...4...4...4..
M20Diag>
```

If no data pattern is specified on the command line, the buffer is filled with the worst-case data pattern for the current data density; that is, the data pattern most likely to result in errors when read back. Example (single density):

```
M20Diag>FD K
Write buffer at $10000-$1007F
M20Diag>MD 10000
00010000  9249 2492 4924 9249  2492 4924 9249 2492     .I$.I$.I$.I$.I$.
M20Diag>
```

Example (double density):

```
M20Diag>FD K
Write buffer at $10000-$1007F
M20Diag>MD 10000
00010000  6DB6 DB6D B6DB 6DB6  DB6D B6DB 6DB6 DB6D     m6[m6[m6[m6[m6[m
M20Diag>
```

11.13 Looping and chaining FD commands

In many cases, the user will want to execute an FD command repeatedly. Also, the user may want to execute several commands with parameters as a group. The LC (Loop Continuous) command and the "!" command line separator allow the user to do this.

For example, suppose the user wants to carry out the following sequence: restore drive, seek to track $4F, write sector 1 with a pattern, write sector 2 with worst case data, read sector 1, read sector 2, and repeat. This sequence would be performed by the following command line:

```
M20Diag>LC FD D!FD E 4F!FD K 1234D6D6!FD H 1!FD K!FD H 2!FD G 1!FD G 2
```

Assuming that a diskette is in the drive, that 128 bytes per sector is selected, and that track $4F has been formatted, entering the above command line would produce a display like this:

```
D Restore..... Running ---> WD1772 status = 10100100 GMX
status = 00011110 PASSED E Seek..... Running ---> WD1772
status = 10100000 GMX status = 00011110 PASSED H Write.....
Running ---> WD1772 status = 10100000 GMX status = 00011110 Write buffer at
$10000-$1007F PASSED H Write..... Running ---> WD1772
status = 10100000 GMX status = 00011110 Write buffer at $10000-$1007F G
Read..... Running ---> WD1772 status = 10100000 GMX
status = 00011110 Read buffer at $20000-$2007F PASSED G
Read..... Running ---> WD1772 status = 10100000 GMX
status = 00011110 Read buffer at $20000-$2007F PASSED ** Pass count = 1 Total
Errors = 00000000000
```

This display will repeat, accumulating passes and errors, until the user enters a BREAK on the console.

11.14 Status returned by FDC test commands

The WD1772 Floppy Disk Controller has an eight bit Status register which indicates the device's current state or the result of the last command executed by the part. Whenever one of the FD tests in 020Bug has the WD1772 execute a command the resulting status is displayed in binary, so the user can examine individual bits conveniently. The GMX Micro-20's CTSR is also displayed; its contents are defined in the System Memory Map included in the GMX Micro-20 hardware documentation. The definitions of the WD1772 status bits are given below.

Bit 0 - BUSY

This bit is 1 while the WD1772 is executing a command. It is 0 when the command is completed.

Bit 1 - Data Request/Index

This bit is a 1 during the execution of read or write commands when a byte is to be transferred in or out of the WD1772. It is a 1 at the end of a command only if the command did not finish normally. If the command is a Restore command this bit is a 1 when the index hole sensor of the selected drive is active.

Bit 2 - Lost Data/Track 00

On a Read, Write, or Write Track command, this bit is a 1 if a byte was not transferred in or out of the WD1772 in time to keep up with the data clock. If the command is a Restore command this bit is the state of the Track 00 sensor of the selected drive.

Bit 3 - CRC Error

This bit is a 1 when a data or address field is read from the diskette and the CRC generated then does not match the corresponding CRC on the diskette. It can be set as a result of Restore, Seek, Read, and Write commands. This bit is a common indicator of soft errors, such as arise from defective diskettes or marginal disk drives.

Bit 4 - Record Not Found

On Read or Write commmands, this bit is a 1 if the WD1772 could not find an address field on the current track which matched the track and sector in the WD1772's Track and Sector registers. On a Restore command, this bit is a 1 if Verify is on and either the Track 00 sensor is not active, or the WD1772 cannot find an address field with a track number of 00. On a Seek command, this bit is a 1 if Verify is on and the WD1772 cannot find an address field with the current track number. The WD1772 reads the current track five times before giving up and setting this flag. If Bit 3 (CRC Error) is also set, the address field was found, but its CRC did not match.

Bit 5 - Record Type/Spin-up

On Read or Write commands, this bit is a 1 if a Deleted Data Address Mark was read or written, Or for a normal Data Address Mark. On Restore and Seek commands it is a 1 if the drive motor spin-up sequence is complete, and the

drive is up to speed.

Bit 6 - Write Protect

On write commands, this bit is a 1 if the Write Protect sensor of the selected disk drive is active, indicating that the Write Protect notch on the diskette is covered.

Bit 7 - Motor on

On any command, this bit is a 1 if the Motor On output of the WD1772 is active.

The FD commands also return status from the GMX Micro-20's Control/Status Register (CTSR). Bits 5, 6, and 7 of this register indicate additional status conditions.

Bit 5 - RDY

This bit indicates the status of the READY signal from the selected drive. If it is returned as a 1, then the drive did not come ready after being selected, i.e., the door is open. This bit may be forced to 0 by jumper JA8 for drives without a Ready output. See p. 8 and p. 11 of the Hardware Technical Manual.

Bit 6 - INT

This bit indicates the status of the INTRQ output of the WD1772. This bit normally is a 1 when a WD1772 command is completed. A level 5 Autovector Interrupt exception will be generated when this bit is set.

Bit 7 - DRQ

This bit indicates the status of the DRQ output of the WD1772. When it is a 1, the WD1772 needs to have a byte of data read from it or written to it. When DRQ interrupts are enabled, a level 7 Autovector Interrupt exception will be generated for each DRQ.

Bits 0 through 4 of this register are set by the sense switch, and not related to the floppy disk interface.

12.0 SASI TESTS

12.1 General Description

This section describes the tests functions available for exercising the GMX Micro-20's SASI port. These tests interact with a controller and hard disk attached to the SASI port to perform I/O and other SASI functions.

TABLE T-6. SASI TESTS

Monitor Command	Title	Section	Page
SA A	Set drive parameters	12.2	D-69
SA B	Scan data lines	12.3	D-72
SA C	Restore	12.4	D-73
SA D	Seek	12.5	D-74
SA E	Read	12.6	D-75
SA F	Write	12.7	D-76
SA G	Compare read and write buffers	12.8	D-77
SA H	Fill write buffer with data	12.9	D-78
SA I	Test interrupt	12.10	D-79
SA J	Park head	12.11	D-80
SA K	Format hard disk	12.12	D-81

12.1.1 Hardware requirements

The SASI tests require a controller and hard disk to be attached to the GMX Micro-20. The controller must be either an OMTI model 20C-1 or a Xebec S1410A. The drive may be any drive model which is compatible with the controller; however, sets of parameters are predefined for five drive models: MiniScribe 3425, Micropolis 1325, Vertex V185, Vertex V170, and Maxtor 1140. The drive must be assigned to the controller's LUN 0.

12.1.2 Hard disk addresses

Nearly all SASI controllers, including the OMTI 20C-1 and the Xebec S1410A, allow the host to treat the disk drive as a continuous block of sequentially numbered sectors starting from 0. These sector numbers are the "logical sector numbers", and are the only address information the host sends to the controller. The controller then converts the logical sector number into a specific cylinder, surface, and sector, based on the drive parameters previously supplied. The sector size jumper on the controller also affects this process, as different numbers of 512 byte and 256 byte sectors fit on a track.

Some of the SASI test commands allow the user to enter an address, but this address is a logical sector number. A user who wishes to access a specific cylinder, surface, and sector must calculate the sector's logical sector number and enter it in hexadecimal.

Example: A media flaw is suspected at cylinder 104, surface 2, of a MiniScribe 3425, which has four surfaces. (Cylinder, surface, and head numbers begin with

0.) The controller is set up for 256 byte sectors, which is 32 sectors per track. The logical sector number of the start of the track is therefore

((104 x 4) + 2) x 32 = 13,376 = $3440

Accessing sectors in the range $3440 to $345F will test for the flawed spot.

12.2 Set drive parameters SA A

This command begins by determining whether a controller is attached attached to the GMX Micro-20's SASI port, and reports whether the controller is an OMTI 20C-1 or a Xebec S1410A. It also reports what model of drive is selected, and displays the current drive parameters. The user may then specify drive parameters by selecting one of the defined drive models or by selecting "other model" and entering parameters individually. If "other model" is selected, the user should read the manufacturer's descriptions of the disk drive and controller installed in the particular system before editing the disk drive parameters.

The parameters which can be edited by the user with the "SA A" command are number of heads, number of cylinders, starting cylinder for write precompensation, starting cylinder for reduced write current, number of sectors per track, fixed or removable drive media, soft or hard sectors, step pulse width, step period, maximum ECC burst length, and sector size. Some of these parameters are defined differently for the OMTI 20C-1 and Xebec S1410A. All but the last of these are set automatically when a defined drive model is selected.

The prompts and definitions for each parameter are given below. All numeric parameters should be entered in decimal, ancd must be terminated by CR. All single letter parameters, including sector size and Xebec step rate, are entered by typing a single valid character, and no CR is needed.

1. Number of heads (1-16).....?

This parameter is the number of separate read/write heads which are used in the drive to access data surfaces. Heads which are used to access servo positioning information only should not be included in this number.

2. Number of cylinders (1-65535).....?

This parameter is the number of data tracks on the surface of a drive platter. Tracks which are contain only servo positioning information, or are reserved as a head parking area should not be included in this number.

3. Write precomp at cylinder (0-1023)....? <OMTI 20C-1>
   Write precomp at cylinder (0-65536)...? <Xebec S1410A>

This parameter is the lowest number cylinder for which the controller should enable write precompensation. Cylinder numbering starts at 0. If write precomp is not used, this parameter should be set to the total number of cylinders on a Xebec S1410A, and to 0 on an OMTI 20C-1.

4. Reduced write at cylinder (0-65536)...? <Xebec S1410A only>

This parameter is the lowest number cylinder for which the controller should enable reduced write current. Cylinder numbering starts at 0. If reduced write is not used, this parameter should be set to the total number of cylinders.

5. Maximum ECC burst length (0-11).....? <Xebec S1410A only>

This parameter is the maximum length of error which the Xebec S1410A is permitted to correct. Any group of error bits which is no longer than this value will be corrected automatically by the Xebec S1410A. This value should be

set to 11 for normal operations, but setting to a smaller value or even 0 is useful when testing for marginal disk drive operation.

6. Sectors/track (1-255, 0=default).....? <OMTI 20C-1 only>

This parameter indicates the number of data sectors to be recorded on each track. If a zero value is entered the controller will use the default value appropriate to the sector size: 32 sectors per track with 256 byte sectors, and 17 sectors per track with 512 byte sectors.

7. Fixed/removable (F or R).....? <OMTI 20C-1 only>

This parameter sets a flag which tells the controller whether the drive has fixed media or a removable cartridge.

8. Soft/hard sector (S or H).....? <OMTI 20C-1 only>

This parameter sets a flag which tells the controller whether the drive media has a fixed sector layout or must be formatted.

9. Step pulse width in usec (0-255).....? <OMTI 20C-1>

This parameter is the length in microseconds of a step pulse sent by the controller to the drive.

10. Step period (N x 50 usec, 0=3.5 usec) ? <OMTI 20C-1>

This parameter is the length in microseconds of the gap between two successive step pulses sent by the controller to the drive. Maximum value for N is 255.

11. Step rate: <Xebec S1410A>

0 = 3 msec unbuffered
4 = 200 usec
5 = 70 usec
6 = 30 usec
7 = 15 usec
8 = 12 usec ?

This parameter is the interval from the beginning of one step pulse sent by the controller to the drive to the beginning of the next.

12. Imbedded servo information (Y/N) .....? <Xebec S1410A>

This parameter sets a flag which tells the controller whether the disk's format includes imbedded pulses of servo control information.

13. Sector size (0=256 1=512) .....?

The sector size parameter (256 or 512 bytes/sector) is independent of drive type. It is set by installing a jumper on the controller board, but cannot be read from the controller by the host. So it must always be entered, even when a defined drive model has been selected.

Example:

```
M20Diag>SA A

OMTI controller installed

Drive model selected: other model
Cylinders = 612  Heads = 4  Sectors/track = 17  Max sector address = $0000A290
Write precomp at cylinder 0
Step pulse width = 2 usec    Step period = 3.5 usec
Fixed disk ------- Soft sectors --- 512 bytes/sector
Drive models:
1 = MiniScribe 3425
2 = Micropolis 1325
3 = Vertex V185
4 = Vertex V170
5 = Maxtor 1140
6 = other model (enter parameters)
Enter 1-6: 6
Number of heads (1-16).................? 4
Number of cylinders (1-65535).........? 612
Write precomp at cylinder (0-1023)....? 0
Sectors/track (1-255, 0=default)......? 0
Fixed/removable (F or R)..............? F
Soft/hard sector (S or H).............? S
Step pulse width in usec (0-255)......? 2
Step period (N x 50 usec, 0=3.5 usec) ? 0
Sector size (0=256 1=512) ............? 1

Drive model selected: other model
Cylinders = 612  Heads = 4  Sectors/track = 17  Max sector address = $0000A290
Write precomp at cylinder 0
Step pulse width = 2 usec    Step period = 3.5 usec
Fixed disk ------- Soft sectors --- 512 bytes/sector

Parameters OK (Y or N) ? Y

M20Diag>
```

12.3 Scan data lines SA B

This command places a rotating data pattern of seven 0s and a 1 on the data lines of the SASI port, which are set to outputs. The test continues until the user hits any key on the console. This command is useful for a technician checking for open or shorted data lines. Example:

```
M20Diag>SA B     <any key terminates>
M20Diag>
```

SA C

This command has the controller perform a Restore command on LUN 0. This causes the drive's head assembly to be moved to the outermost track at a slow, unbuffered stepping rate. Example:

```
M20Diag>SA C
C Restore..................... Running ---> PASSED
M20Diag>
```

If the controller cannot restore the drive, it will return an error to the host, which is displayed; see section 12.13 for the explanation of SASI error reporting.

12.5 Seek SA D

SA D [<sector>]

This command performs a seek to the specified sector. The target sector may be specified on the command line; if it is not the user will be prompted for it. Example:

```
M20Diag>SA D 1234
D Seek ........................... Running ---> PASSED
M20Diag>
```

or

```
M20Diag>SA D
D Seek ........................... Running ---> Sector number? $1234
PASSED
M20Diag>
```

If the sector number is out for range for the specified drive model, a message will be printed and the prompt issued again. Example:

```
M20Diag>SA D
D Seek ........................... Running ---> Sector number? $123467
Sector  too large
Sector number? $1234
PASSED
M20Diag>
```

If the controller cannot seek the drive to the specified sector, it will return an error to the host, which is displayed; see section 12.13 for the explanation of SASI error reporting.

12.6 Read SA E

SA E [<sector>]

This command performs a read of the specified sector. The read data buffer is at $20000. The target sector may be specified on the command line; if it is not the user will be prompted for it. Example:

```
M20Diag>SA E 1234
E Read ......................... Running ---> Read buffer at $20000
PASSED
M20Diag>
```

or

```
M20Diag>SA E
E Read ......................... Running ---> Sector number? $1234
Read buffer at $20000
PASSED
M20Diag>
```

If the controller cannot read the specified sector, it will return an error to the host, which is displayed; see section 12.13 for the explanation of SASI error reporting.

12.7 Write SA F

SA F [<sector>]

This command performs a write to the specified sector. The write data buffer is at $10000. The target sector may be specified on the command line; if it is not the user will be prompted for it. Example:

```
M20Diag>SA F 1234
E Read ........................... Running ---> PASSED
M20Diag>
```

or

```
M20Diag>SA F
F Write .......................... Running ---> Sector number? $1234
PASSED
M20Diag>
```

If the controller cannot write the specified sector, it will return an error to the host, which is displayed; see section 12.13 for the explanation of SASI error reporting.

12.8 Compare read and write buffers SA G

SA G

This command performs a byte by byte comparison of the data in the read and write buffers. Example:

```
M20Diag>SA G
G Compare read and write buffers Running ---> PASSED
M20Diag>
```

If any differences are found, they are reported and an error is recorded. Example:

```
M20Diag>SA G
G Compare read and write buffers Running --->
Data error at byte 0B1 - data written = 09 - data read = 00
Data error at byte 0F5 - data written = D2 - data read = 48..... FAILED
M20Diag>
```

12.9 Fill write buffer with data SA H

SA H [<pattern>]

This command fills the write buffer with a specified longword data pattern. The write buffer is at $10000. Example:

```
M20Diag>SA H 1234ACE0
M20Diag>MD 10000
00010000 1234  ACE0  1234  ACE0   1234  ACE0  1234  ACE0      .4,`.4,`.4,`.4,`
M20Diag>
```

If no data pattern is specified on the command line, then the buffer is filled with the worst case data pattern, that is the data pattern most likely to cause read errors. Example:

```
M20Diag>SA H
M20Diag>MD 10000
00010000 6DB6  DB6D  B6DB  6DB6   DB6D  B6DB  6DB6  DB6D      m6[m6[m6[m6[m6[m
M20Diag>
```

12.10 Test interrupt SA I

SA I

This command tests the interrupt generation function of the SASI port. The port can produce a level 1 Autovector Interrupt; this command enables the interrupt and performs a controller command which should generate this interrpt. Example:

```
M20Diag>SA I
I Test Interrupt..... Running ---> PASSED
M20Diag>
```

If the interrupt does not occur, an error is reported. Example:

```
M20Diag>SA I
I Test Interrupt..... Running ---> .....FAILED
M20Diag>
```

12.11 Park head SA J

SA J

When a system containing a hard disk drive is moved or shipped, a jolt or bump may cause the head assembly to bang against the recording surfaces, causing media diefects and loss of recorded data. To prevent this, disk drive manufacturers designate a cylinder outside the normal data storage area as a "landing zone", where the heads may be "parked" without endangering the useful parts of the disk. This command allows the user to move the drive's head assembly to the park area for shipping or other movement. Drives which have a voice coil positioning cannot be parked by the user: they park themselves automatically.

**** WARNING!!!! ****

Trying to park a voice coil drive will overwrite a data track, probably track 00.

The complete procedure for parking a drive is as follows:

First, the user is prompted for and selects the Logical Unit Number of the drive to be parked.

Second, the user is prompted for and selects the sector size the controller is set for.

Third, the user is prompted for and enters the cylinder number of the park area. This number should be obtained from the manufacturer's description of the drive being parked. This number must be correct; if it is wrong a track containing data (possibly track 00) will be overwritten.

Fourth, the user is prompted for two confirming responses to ensure that the cylinder number is correct. Then the drive is restored to cylinder 00.

Last, the user is prompted for a final confirming response, and then the drive is parked. Example:

```
M20Diag>SA J

OMTI controller installed
Which LUN to park (0/1) ? 0
Sector size (0=256 1=512).............? 1
Cylinder number (decimal) in park area? 629
Park head at cylinder 0629 (Y/N) ? Y
Reenter cylinder number to confirm: 629


        ***** WARNING *****

Parking at the wrong address will destroy data!
This could lose everything on the drive!
The confirm code is 21386.
Enter this number to verify the cylinder again: 21386
Drive is parked
M20Diag>
```

SA K

This command allows the user to format an entire hard disk drive. This command should be used with great caution, as all data on the reformatted drive will be lost. The procedure for formatting a drive with the SA K command is as follows:

First, the current selected drive model and parameters are reported. If the parameters are wrong, the drive will not be formatted correctly.

Second, the user is prompted for and selects the Logical Unit Number of the drive to be formatted.

Third, the user is prompted for three further confirming responses.

Then the drive is formatted. Example:

```
M20Diag>SA K


Drive model selected: MiniScribe 3425
Cylinders = 612  Heads = 4  Sectors/track = 17  Max sector address = $0000A290
Write precomp at cylinder 0
Step pulse width = 2 usec    Step period = 3.5 usec
Fixed disk ------- Soft sectors --- 512 bytes/sector
Which LUN to format (0/1) ? 0
Are you sure (Y/N) ? Y
Enter LUN again: 0


        ***** WARNING *****

Formatting the drive will destroy all data on it!
Make sure you have the correct drive, model, and LUN!
The confirm code is 7968.
Enter this number to verify the cylinder again: 7968
Now formatting drive.... ....Done
M20Diag>
```

12.13 Looping and chaining SA commands

In many cases, the user will want to execute an SA command repeatedly. Also, the user may want to execute several commands with parameters as a group. The LC (Loop Continuous) command and the "!" command line separator allow the user to do this.

For example, suppose the user wants to carry out the following sequence: restore drive, write sector $4201 with worst case data, write sector $BCDE with a pattern, read sector $4201, read sector $BCDE, and repeat. This sequence would be performed by the following command line:

```
M20Diag>LC SA C!SA H!SA F 4201!SA H 3232999!SA F 9BDF!SA E 4201!SA E BCDE
```

Assuming that a comtroller and formatted drive are connected, entering the above command line would produce a display like this:

```
C  Restore.....  Running ---> PASSED Write buffer at $10000 F
Write.....  Running ---> PASSED Write  buffer  at  $10000  F
Write.....  Running  --->  PASSED  E
Read.....  Running ---> Read  buffer  at  $20000  PASSED  E
Read.....  Running  --->  Read  buffer at $20000 PASSED **
Pass count = 1   Total Errors = 00000000000
```

This display will repeat, accumulating passes and errors, until the user enters a BREAK on the console. When an errors is detected, error information will be reported in the display.

12.13 SASI controller error reporting

Many of the SASI test commands invoke controller commands. If a controller command terminates with an error, four bytes of error information is returned to the host. This error information is displayed by the SASI test commands. Example:

```
M20Diag>SA C
C Restore..................... Running --->
Controller error: status = 21000001
.....FAILED
M20Diag>SA C
```

Consult the manufacturer's manual for a complete explanation of this error status information.

12.14 SASI handshake error reporting

Execution of a controller command requires the exchange of several bytes of information between the host and controller, and the handshake logic of the SASI port functions to keep these exchanges in order. There are five handshake signals from the controller to the host; the GMX Micro-20 has a special circuit which converts these signals into a BUSY bit and four status bits in the SASI Status Register (SASR), which is described in the System Memory Map in the GMX Micro-20 Hardware Technical Manual. Each stage in the performance of a controller command is indicated by the setting of one and only one of these four bits. These steps are checked by the test commands, and if an error is detected, it is recorded and reported. Example:

```
M20Diag>SA C
C Restore..................... Running --->
Testing for SCMD - SASR = 11110000   Bit not set in SASR
.....FAILED
```

The handshake checker reports if a bit is not set when it should be, and if a bit is set when it shouldn't be, and if the BUSY bit is cleared before the command is finished.

13.0 ARCnet INTERFACE TESTS

13.1 General Description

This section describes the tests available for testing a GMX SBC-AN ARCnet network interface board plugged into the GMX Micro-20's I/O expansion port. If no SBC-AN is installed, these tests will report "No ARCnet interface installed". These tests check out some basic functions of the board's hardware and connection to the GMX Micro-20. No testing of network operation is performed, so no connection to any outside equipment is required.

For a complete explanation of the SBC-AN hardware and functions, refer to the "GMX SBC-AN ARCnet Interface Board User's Manual".

TABLE T-6. ARCnet INTERFACE TESTS

Monitor Command	Title	Section	Page
AN A	ARCnet wake-up test	13.2	D-85
AN B	ARCnet DIP-switch test	13.3	D-86
AN C	ARCnet interrupts test	13.4	D-87
AN D	ARCnet buffer test	13.5	D-89

13.1 ARCnet wake-up test AN A

13.2.1 Description

After power-up or RESET the SBC-AN should go to a specific state. The control register of the COM 9026 network controller should read $95, and the value $D1 should be stored in the first byte of the on-board buffer RAM. Both of these conditions are checked.

13.2.1 Command Input

```
M20Diag>AN A
```

13.2.2 Response/Messages

After the command is entered, the display should appear as follows:

```
A ARCnet wake-up test..... Running --->
```

If the wake-up data is not correct, then the values found will be displayed along with the values expected, as follows:

```
A ARCnet wake-up test..... Running --->
Status byte+wake-up byte =$6789, not $95D1
..... FAILED
```

If no errors occur, the display will appear as follows:

```
A ARCnet wake-up test.....Running ---> PASSED
```

If no SBC-AN is installed, the display will appear as follows:

```
A ARCnet wake-up test..... Running --->  No ARCnet interface installed
PASSED
```

13.3 ARCnet DIP-switch test AN B

13.3.1 Description

The COM 9026 network controller uses the value of the DIP-switch bank on the SBC-AN as its node ID in network operations. At power-up or RESET, the COM 9026 reads the switches and stores the value in the second byte of the on-board buffer RAM. This test causes a reset of the SBC-AN, waits for the COM 9026 to complete its initialization, then reads the DIP-switch value from the buffer RAM and displays it in binary. This process is repeated until the user enters a BREAK on the console. No line feed is issued between displays, so the output is rewritten on the same line of the screen. If the user changes a switch, the display will be updated immediately. No errors are possible.

13.3.2 Command Input

```
M20Diag>AN B
```

13.3.3 Response/Messages

After the command is entered, the display should appear as follows:

```
DIP switches on ARCnet board = 00001000
```

This display will continue until terminated by BREAK. If no SBC-AN is installed, the display will appear as follows:

```
No ARCnet interface installed
```

13.4 ARCnet interrupts test AN C

13.4.1 Description

This test exercises the interrupt generation function of the SBC-AN. When the COM 9026 network controller issues an interrupt, the SBC-AN produces a level 3 or level 4 autovector interrupt to the 68020. The interrupt level is selected by an on-board jumper, and interrupt output is enabled or disabled by a bit in the SBC-AN control register. At RESET, the COM 9026 generates a POR interrupt, which is used in this test.

13.4.1 Command Input

```
M20Diag>AN C
```

13.4.2 Response/Messages

After the command is entered, the display should appear as follows:

```
C ARCnet interrupts test..... Running --->
```

If a level 3 or 4 interrupt is generated, the interrupt level is reported as follows:

```
C ARCnet interrupts test..... Running --->
Level 3 interrupt received
PASSED
```

If any other interrupt is received, the test will report an error as follows:

```
C ARCnet interrupts test..... Running --->
ERROR - Level 5 interrupt received
..... FAILED
```

If no interrupt is received, the test will report an error as follows:

```
C ARCnet interrupts test..... Running --->
ERROR - no interrupt generated
..... FAILED
```

If an interrupt is received after the POR flag in the COM 9026 is cleared, the test will report an error as follows:

```
C ARCnet interrupt test..... Running --->
ERROR - interrupt repeated after POR was cleared
..... FAILED
```

If an interrupt is received while interrupt generation is disabled, the test will report an error as follows:

```
C ARCnet interrupts test..... Running --->
ERROR - interrupt was not properly masked
..... FAILED
```

If a spurious interrupt is generated, an the test will report an error as follows:

```
C ARCnet interrupts test..... Running --->
ERROR - spurious interrupt
..... FAILED
```

If no SBC-AN is installed, the display will appear as follows:

```
C ARCnet interrupts test..... Running ---> No ARCnet interface installed
PASSED
```

13.5 ARCnet buffer test AN D

13.5.1 Description

This test checks out the 2 Kbyte RAM buffer on the SBC-AN. The buffer is used for storage of incoming and outgoing network messages. It is accessible to the 68020 in four pages of 512 bytes, selected by two bits in the SBC-AN control register. This test invokes five of the tests in the memory test package for each page, with the bounds of the accessable buffer page as the bounds of the test area. Errors are reported as in section 6.13, "Description of Memory Error Display Format". The test also checks for correct decoding of the page select bits.

13.5.1 Command Input

```
M20Diag>AN D
```

13.5.2 Response/Messages

After the command is entered, the display should appear as follows:

```
D ARCnet buffer test..... Running --->
Now testing page 0 of the ARCnet buffer
 - Random Inversion Test
 - March Address Test
 - Walk a bit Test
 - Random Byte Test
 - TAS Test
<repeated for pages 1, 2, and 3>
```

If no SBC-AN is installed, the display will appear as follows:

```
D ARCnet buffer test..... Running ---> No ARCnet interface installed
PASSED
```

If errors are detected in a page, they are reported as described in section 6.13. If an error is detected in page decoding, it is reported as follows:

```
D ARCnet buffer test..... Running --->
Now testing page 0...
Now testing page 1...
Now testing page 2...
Now testing page 3...
ERROR in buffer paging - Page 0 written  Page 1 modified
..... FAILED
```

14.0 PARALLEL I/O EXPANSION BOARD TESTS

14.1 General Description

This section describes the functions available for exercising an SBC-60P Parallel I/O Board attached to the GMX Micro-20's I/O Expansion Connector.

TABLE T-7. PARALLEL EXPANSION BOARD TESTS

Monitor Command	Title	Section	Page
PX A	Data, handshake, and IRQ test	14.2	D-93
PX B	P4 connector test	14.3	D-94
PX C	Data and handshake toggle	14.4	D-95

14.1.1 Hardware requirements

The tests described in this section will work only if an SBC-60P Parallel I/O Board is installed in the GMX Micro-20's I/O Expansion Port. The address of the SBC-60P may be jumpered either to $00FF9000 or to $00FF9080.

14.1.1.1 Requirements for PX A

For test PX A, a set of dummy plugs must be installed in connectors P1, P2, and P3. Each of these plugs makes a loopback connection of the Port A data lines to the Port B data lines of the corresponding 68230 PI/T. The loopback path includes the external buffers for each port. The PI/T's handshake lines are also looped back: H1 to H2, and H3 to H4. To make these loopback paths, the dummy plug for each connector must connect certain pins, as shown below.

Pin 15	(H1)	to	Pin 14	(H2)
Pin 11	(H3)	to	Pin 9	(H4)
Pin 47	(PA0)	to	Pin 31	(PB0)
Pin 45	(PA1)	to	Pin 29	(PB1)
Pin 43	(PA2)	to	Pin 27	(PB2)
Pin 41	(PA3)	to	Pin 25	(PB3)
Pin 39	(PA4)	to	Pin 23	(PB4)
Pin 37	(PA5)	to	Pin 21	(PB5)
Pin 35	(PA6)	to	Pin 19	(PB6)
Pin 33	(PA7)	to	Pin 17	(PB7)

Jumper blocks JA1, JA3, and JA5 determine control of the switching of the external bidirectional data buffers on Ports A and B of the PI/Ts. Test PX A requires control of the direction by the PC0 and PC1 outputs of each PI/T, so these jumpers must be set as shown below.

JA3-20	(PI/T A PC0)	to	JA3-19	(Port A buffer pin 1)
JA3-23	(PI/T A PC1)	to	JA3-24	(Port B buffer pin 1)
JA5-4	(PI/T B PC0)	to	JA5-3	(Port A buffer pin 1)
JA5-7	(PI/T B PC1)	to	JA5-6	(Port B buffer pin 1)
JA1-6	(PI/T C PC0)	to	JA1-5	(Port A buffer pin 1)
JA1-9	(PI/T B PC1)	to	JA1-10	(Port B buffer pin 1)

Data lines PA0 through PA3 of PI/T A may be individually jumper configured to function in reverse of the direction selected at pin 1 of the Port A external buffer. Jumper block JA3 controls this option, which must not be selected, so these jumpers must be set as shown below.

Pin 16	(PI/T A PA0)	to	Pin 14	(Buffer pin 18)
Pin 12	(PI/T A PA1)	to	Pin 10	(Buffer pin 17)
Pin 8	(PI/T A PA2)	to	Pin 6	(Buffer pin 16)
Pin 4	(PI/T A PA3)	to	Pin 2	(Buffer pin 15)

Pin 15	(Buffer pin 2)	to	Pin 13	(connector P2 pin 47)
Pin 11	(Buffer pin 3)	to	Pin 9	(connector P2 pin 45)
Pin 7	(Buffer pin 4)	to	Pin 5	(connector P2 pin 43)
Pin 3	(Buffer pin 5)	to	Pin 1	(connector P2 pin 41)

The handshake lines of PI/T B are wired directly to Pins 9 through 14 of connector P3. However, those of PI/T A and PI/T C are not wired directly to connectors P2 and P1. These lines are routed through jumper blocks JA7 and JA6 respectively. To complete the loopback of the handshake lines, connections must be made in each jumper block as shown below.

Pin 22	(H1)	to	Pin 23	(connector pin 15)
Pin 19	(H2)	to	Pin 20	(connector pin 14)
Pin 16	(H3)	to	Pin 17	(connector pin 11)
Pin 13	(H4)	to	Pin 14	(connector pin 9)

The H2 and H4 handshake lines of all three PI/Ts pass through buffers which must be jumper selected for output. This is done by connecting pin 1 to pin 3 and pin 2 to pin 4 in jumper blocks JA8 through JA13.

Test PX A also requires the PC5/PIRQ output of the PI/Ts to be configured to generate level 3 Autovector Interrupts, and PC3/TOUT to be configured to generate level 4 Autovector Interrupts. For this configuration, jumper block JA18 must be set as shown below.

Pin 5	(PC5/PIRQ)	to	Pin 6	(Pin 32 of I/O Expansion Port)
Pin 3	(PC3/TOUT)	to	Pin 4	(Pin 30 of I/O Expansion Port)

14.1.1.2 Requirements for PX B

For test PX B, a dummy plug and cable set must be installed in connectors P4 and P3. This test rig must loop back the Port A and Port B data lines of PI/T C as for test PX A. Additionally, three of the Port C data lines of PI/T C (PC6, PC4, and PC7) must be connected to the PC2/TIN lines of the three PI/Ts, as shown below.

P4 pin 23	(PI/T C PC6)	to	P3 pin 3	(PI/T A PC2/TIN)
P4 pin 21	(PI/T C PC4)	to	P3 pin 5	(PI/T B PC2/TIN)
P4 pin 31	(PI/T C PC7)	to	P3 pin 7	(PI/T C PC2/TIN)

The paths from connector P3 to the PC2/TIN lines of the PI/Ts are interrupted by jumper block JA17. The connections must be completed by setting these jumpers as shown below.

Pin 1	(PI/T C	PC2/TIN)	to	Pin 2	(P3 pin 7)
Pin 3	(PI/T A	PC2/TIN)	to	Pin 4	(P3 pin 5)
Pin 5	(PI/T B	PC2/TIN)	to	Pin 6	(P3 pin 3)

Test PX B also requires the buffer direction control jumper blocks (JA3, JA5, and JA7) to be configured the same as for test PX A.

14.2 Data, handshake, and IRQ test PX A

This command exercises most of the external connections of the three PI/Ts. All of the Port A and Port B data lines are tested, the handshake lines are tested, and both of the IRQ outputs are tested. The board must be configured as described in section 14.1.1.1.

14.2.2 Command Input

```
M20Diag>PX B
```

14.2.3 Response/Messages

After the command is entered, the display should appear as follows:

```
A Data, handshake, and IRQ test. Running --->
```

If defective data lines are detected, the error will be reported as follows:

```
A Data, handshake, and IRQ test. Running --->
Error on PI/T A  --  Data lines  --  error mask = 01100001
...Failed
```

In the error mask, 1s represent malfunctioning data lines. Because each path includes both the Port A and Port B data lines, a fault on either side produces the same message.

If a handshake line fails to operate, the error will be reported as follows:

```
A Data, handshake, and IRQ test. Running --->
Error on PI/T B  --  Port A handshake [H1-H2]
...Failed
```

Because each path includes two handshake lines, a fault in either line produces the same message.

The PC5/PIRQ output should produce a level 3 Autovector Interrupt, and the PC3/TOUT output should produce level 4 Autovector Interrupt. If one of these interrupt outputs malfunctions, the test will report no interrupt:

```
A Data, handshake, and IRQ test. Running --->
Error on PI/T A  --  No interrupt from PIRQ [PC5]
...Failed
```

or wrong IRQ:

```
A Data, handshake, and IRQ test. Running --->
Error on PI/T A  --  Wrong IRQ - level 5 AV from PIRQ [PC5]
...Failed
```

or spurious interrupt:

```
A Data, handshake, and IRQ test. Running --->
Error on PI/T A  --  Spurious interrupt from TOUT [PC3]
...Failed
```

14.3 P4 connector test PX B

This test exercises the PI/T C inputs and outputs through connector P4. These lines include the Port A and Port B data lines, and three lines of Port C. These last are connected to the PC2/TIN lines of the three PI/Ts, which are also tested. The board must be configured as described in section 14.1.1.2.

14.3.2 Command Input

```
M20Diag>PX B
```

14.3.3 Response/Messages

After the command is entered, the display should appear as follows:

```
A P4 connector test..... Running --->
```

If defective data lines are detected, the error will be reported as in test PX A. If a PCn-to-PC2/TIN path fails to operate, the error will be reported as follows.

```
A P4 connector test..... Running --->
Error on PI/T A  --  PC2/TIN stuck
```

Because each of these paths includes two lines, a fault on either line will produce the same error message.

14.4 Data and handshake toggle PX C

This command causes the Port A data lines, Port B data lines, and handshake outputs (H2 and H4) of all three PI/Ts to toggle between high and low states, which is useful to a technician checking for shorted or coupled lines. This toggling is rapid (each line should change state about once every 100 usec), and pseudo-random (to avoid coupling). Once started, the command will run indefinitely, but can be terminated by BREAK.

WARNING: when this command is started, the dummy plugs and cables used with tests PX A and PX B must not be in place. If any such plugs or cables are installed when PX C is run, this may result in output buffers with conflicting levels being connected, which could damage one or both buffers.

14.4.2 Command Input

```
M20Diag>PX C
```

14.4.3 Response/Messages

After the command is entered, the display should appear as follows:

```
M20Diag>PX C
Now toggling...
```

13 -2  days in 2yrs

20 1 

14 0