MICROPAC 180
MTBASIC

USER'S GUIDE

REVISION 1.6

EMAC, inc.

EQUIPMENT MONITOR AND CONTROL
CARBONDALE, IL 62901
618-529-4525

Table of Contents

Chapter 1: Introduction to MTBASIC

MTBASIC is a high speed interactive multitasking BASIC compiler. Since most computers are not equipped for true multitasking, MTBASIC provides the software resources required for this feature.

MTBASIC is a completely interactive compiler. This is an almost unheard of concept. Traditional compilers require the user to first edit a source code file, then compile the source code file into an intermediate object code file, link the intermediate object file into an absolute object file, load the absolute object file into memory and finally execute the compiled code. This becomes unbearably tedious while debugging, since it may take minutes to make a single change to the program.

MTBASIC functions much like an interpreter. The BASIC program is entered with line numbers which determine the order the statements appear in the program. The program is compiled and run by typing the direct command RUN. In an interpreter, the RUN command simply starts interpretation of the high level BASIC language program. In MTBASIC, however, RUN first completely compiles the program into object code, then executes the program. The compilation is very quick; typically MTBASIC compiles over 100 BASIC statements per second.

MTBASIC runs under two different environments. A traditional compiler uses only a single environment where the compiled code is always executed independently of any true interaction with the operator. With MTBASIC, during the program development phase the operator interfaces to the compiler exactly as he would to an interpreter. When the program is finally debugged, the compiled code can be written to an EPROM for execution in a stand-alone mode. The interpreter-like interaction is designed for friendly, easy debugging of programs, and the stand-alone mode is provided for finished applications.

One of the major features of MTBASIC is multitasking. By definition, multitasking is the ability of a system to run several activities ("tasks") concurrently. A single processor computer cannot really run more than one task at any specific instant. With concurrent processing, although only one task is in control of the processor at any one time, the software within the compiler automatically switches processor time between each of the various tasks, so, to the operator, it appears that all tasks are running simultaneously. MTBASIC provides all of the logic needed to support multitasking. Only a few statements need to be issued by the user within the program. Multitasking is important in real time processing, where individual tasks may be assigned responsibility for handling different processes. Individual tasks may also be assigned to particular devices.

MTBASIC also supports windowing. The user can partition the terminal screen into up to 10 distinct areas, each of which can receive output from a separate task. Also, any task can access any window, so pop up menus are simple to implement. Note that MTBASIC must be configured for the user's terminal (see Chapter 8) before windowing commands are used.

Finally, it is important to note the constraints under which MTBASIC operates. EMAC has made no attempt to make MTBASIC compatible with any other BASIC because MTBASIC has many special features. For example, most compilers are not resident in memory during execution of the user's program. Even the compiler and the object code are not co-resident, since compilers usually write object code to disk as it is generated. MTBASIC, on the other hand, keeps the entire compiler, the runtime package, the entered source code, and the generated object code in memory during the debugging process. In order to minimize the amount of memory needed for the compiler, EMAC has chosen to include only the BASIC commands that are most useful for imbedded control applications.

GETTING STARTED

This section is a step by step guide to getting started with MTBASIC and entering and running a simple MTBASIC program.

The default communication port COM1 does not require any form of handshaking to operate, although it makes available handshaking lines (handshake out on pin 4 and handshake in on pin 6). It is left up to the user to provide drivers in order to make use of these lines. Some terminals, however, require handshaking to operate, in this case EMAC recommends the use of a null modem cable (refer to the manual that accompanied your terminal to construct a null modem cable). The recommended cable configurations are shown below. "Handshaking" lines are not required by the MICROPAC 180 but may be necessary for the IBM PC and compatibles used as terminal emulators. If the terminal emulator requires handshaking lines, wire a null modem cable by connecting together the PC's RS-232 handshake lines CTS, DSR, DCD and DTR. The diagrams below show the required connections for PCs and terminals with DB25 connectors and PCs with DB9 connectors.

```
            PC or ADM
            and WYSE
MICROPAC    TERMINALS

HDR3 PINS   DB25 PINS
TX   3 ———  3   RX
RX   5 ———  2   TX
GND  9 ———  7   GND

(optional null-modem connections)
           ┌─  5   CTS
           ├─  6   DSR
           ├─  8   DCD
           └─  20  DTR
```

```
MICROPAC    PC

HDR3 PINS   DB9  PINS
TX   3   ———  2   RX
RX   5   ———  3   TX
GND  9   ———  5   GND

(optional null-modem connections)
           ┌─  8   CTS
           ├─  6   DSR
           ├─  1   DCD
           └─  4   DTR
```

Note: When a data terminal has 2 connectors, use the one marked "MODEM".

Configure your terminal to the following parameters: 9600 baud, no parity, one stop bit, and 8 data bits

1. To begin, first apply power to the board. Your responses will be in bold face, the computer's responses will be in plain face. If the logon message below does not appear, see the troubleshooting section of this manual.

```
EMAC Inc.
MTBASIC Compiler
Copyright 1990-93 Softaid/EMAC, Inc.
Version 1.0  License No.  00

>
```

2. Before using windowing or the example programs on RAMDISK (if you have the optional RAMDISK), you must first configure MTBASIC for your particular terminal (Chapter 8 explains how to configure your terminal). The example given in this section does not require terminal configuration.

3. At this point, if there was another program in memory, you would need to use the direct command NEW before beginning a new program. Since, in this example, there is no program currently in memory, you can just begin to enter a new program (press return after typing each line).

```
>5 STRING NAME$
>10 PRINT " This is the example program."
>20 PRINT "It will print a simple message."
>30 PRINT "What is your name?"
>40 INPUT NAME$
>50 PRINT "Hello, ",NAME$
```

4. This is the example program. The next command will compile and run it.

```
>RUN

COMPILED
This is the example program.
It will print a simple message.
What is your name?
Sam
```

```
Hello,      Sam
```

5. There is an error in line 50. The second comma simulates a tab after "Hello,". To change line 50, type the following:

```
> 50 PRINT "Hello, "; NAME$
```

6. To verify that the line has been changed, type:

```
> LIST 50

50 PRINT "Hello, "; NAME$
```

7. COMPILE the program first and then run it by entering GO.

```
>COMPILE
COMPILED

>GO
This is the example program.
It prints a simple message.
What is your name?
Sam
Hello, Sam
```

The following steps require the optional RAMDISK.

8. The next step is to save the program to RAMDISK. The SAVE command will save the program as an ASCII file.

```
>SAVE PROG
>
```

9. Now clear the program from memory and verify that it is gone. The LIST command will show that no program exists.

```
>NEW
>LIST
>
```

10. If you want to run PROG again, enter:

```
>LOAD PROG
```

11. You can type RUN to start the program or, to view the program, enter:

```
>LIST

5 STRING NAME$
10 PRINT " This is the example program."
20 PRINT "It will print a simple message."
30 PRINT "What is your name?"
40 INPUT NAME$
50 PRINT "Hello, ";NAME$
```

To run this program now, see step 4.

There are sections in the manual that explain, in detail, all of the commands and statements used in this example.

LANGUAGE RESTRICTIONS

MTBASIC has a few restrictions which are important for the user to note. Some of these are common to other BASIC implementations while others are peculiar to MTBASIC.

1. All variables used in each program must be defined with INTEGER, STRING, or REAL statements. These declarations must all be made before the first executable statement in the program.

2. No distinction is drawn between variables by mode. This means that the variable A2 is the same variable as A2$. If the variable is declared as both integer and string (both INTEGER A2 and STRING A2$ statements exist), a compilation error will result. Similarly, variable A is the same as dimensioned variable A(n).

3. MTBASIC supports a maximum of two dimensions for real and integer variables. MTBASIC allows a maximum of 255 variables in one program. Any variable name may be used that fits the rules given in Chapter 3; however, no more than 255 variables may be declared.

4. Subscripted variables may not be used as the index in a FOR/NEXT loop. This prevents the user from unintentionally altering the subscript of the subscripted variable, causing mysterious failures of the loop.

5. There is no DIM statement. Instead, since all variables must be declared, dimensions are declared within the INTEGER, STRING and REAL statements themselves.

6. All DATA statements must be given prior to a READ statement. Below are correct and incorrect ways to arrange DATA statements.

```
 WRONG                                      RIGHT
10 INTEGER A,B,C,D,E,F              10 INTEGER A,B,C,D,E,F
20 DATA 11,22,3                     20 DATA 11,22,3
30 PRINT "A COMMAND"                30 PRINT "A COMMAND"
30 READ A,B,C                       30 DATA 5,6,3
40 DATA 5,6,3                       40 READ A,B,C
50 READ D,E,F                       50 READ D,E,F
```

7. Different Tasks may not call common subroutines. If this must be done, use user defined functions

Chapter 2: Direct Commands

Direct commands are those commands given to MTBASIC to control its operation. Direct commands are not part of MTBASIC programs. These are the MTBASIC direct commands:

BURN	Causes EPROM programmer to store data on EPROM
CLS	Clears the screen
COMPILE	Compiles a program without running it
CONSOLE	Directs output to console (disables PRINTER)
DEL	Deletes a file
DIR	Shows the directory of the RAMDISK
DUMP	Shows the contents of memory
EDIT	Allows editing of a line of BASIC
END	Marks the end of a source program file
ERASECHK	Tells whether an EPROM is erased or not
ERROR	Turns on runtime error checking (default is ON)
FORMAT	Prepares RAM to be used as RAMDISK
GO	Starts an already compiled program running
IN	Shows the contents of an input port in hex
LIST	Displays the program code
LISTT	Displays the program, pausing after a full screen
LOAD	Reads a source file from RAMDISK
NEW	Erases the current program
NOERR	Turns off runtime error checking
OUT	Sends a byte to the selected output port
PRN	Sends output to the printer (disables CONSOLE)
RCOMPILE	Compiles a ROMable version of a program
RECEIVE	Writes data from the console to a RAMDISK file
RHEX	Sends an Intel hex file to the console
RUN	Compiles and runs a program
SAVE	Saves a program's source code to RAMDISK
STATUS	Displays the amount of available memory
TBURN	Makes a ROMed version of a program
TYPE	Sends a RAMDISK file to the console
VERIFY	Verifies that EPROM data was correctly burnt

BURN <source>,<dest>,<qty>,<type>

This command supports the EPROM programmer board (part number E020-08). It copies <qty> bytes from system address <source> to <dest> address of the EPROM described by the value of <type>. The value for <source> must be greater than $8000. See the chapter on stand alone applications for more information on making EPROM based MTBASIC programs.

CLS

This clears the terminal screen.

COMPILE

COMPILE causes MTBASIC to convert the program currently in memory to machine language. COMPILE is identical to the RUN direct command, except that the compiled program is not executed. The COMPILEd program may be executed via the GO command. COMPILE will compile a program without producing error checking code if a NOERR command has been issued, otherwise if an ERROR command has been issued or if the board has just been turned on, a program will be compiled with error checking code.

COMPILE B

This command is useful when you are writing a very large program and need to optimize the compile space usage. When the COMPILE command is used to compile a program that has more than one task and that program runs out of space, the program will compile again automatically into banks from the point where it ran out of compile space. The problem with this is that all compile information prior to this point is not displayed. The COMPILE B forces the program to be compiled in banks from the start allowing you to see all the compile information regarding the size of each task and the bank which the tasks reside. This information allows you to reorganize you program to fit the most tasks in a bank as possible.

Before giving the guidelines for writing a bank compiled program some definitions will be given. All the code before the first TASK statement is defined as the lead task, or task 0. The following tasks are now defined as starting with the TASK statement and ending with the statement before the next TASK statement or, if it is the last task, the last statement of the program.

To allow tasks to be in other banks, there is a restriction that a task's transfers (via GOTO, GOSUB, etc.) can only be to line numbers within itself (consequently, the lead task cannot transfer to another task). Otherwise an illegal transfer error will occur. For this reason, in a program to be bank compiled, all tasks should be organized consecutively, with all subroutines used by a task residing within that task.

Subroutines that are used by more than one task, should be rewritten as user defined functions. For example, a subroutine which is rewritten as a function called CHROUT, can be called with the statement GOSUB CHROUT (see Functions and User Defined Functions in the MTBASIC manual). Note that user defined functions reserve access to only one task at a time, so the function must execute to completion before another task can access it. This may cause problems in time critical applications, in which case it may be useful to duplicate the subroutine into each task that uses it.

The bank compiling messages are as follows:

Task <task#> Bank <bank#> Size <size>

This indicates that task number <task#> has just been compiled and that it resides in memory bank <bank#> and occupies <size> bytes of the memory bank. This allows you to organize your tasks if you need to optimize bank space usage.

<line#> → <train#> Illegal Transfer Error

This indicates that a transfer (i.e. GOTO or GOSUB) was made to a line outside of the current task. The line number being transferred to is <train#> and <line#> is the line number where the error occurred. If you receive the message without the line numbers, the error occurred before the bank compiling messages were enabled. Use the "B" option when compiling to find the line number(s) of the transfer error(s).

Task <task#>
<line#> Not Enough Memory to Compile Program

This occurs when a task (even the lead task) is too big. You will need to break down the task specified by <task#> into one or more other tasks or change portions of the task to functions and put them in the lead task (task 0), if there is room. The error will also occur if the compiler runs out of banks, but in this case **Task <task#>** is not shown.

CONSOLE

CONSOLE causes output previously directed to the printer by the PRN command to be directed back to the system console.

DEL <filename>

This deletes <filename> from the RAMDISK and gives a error message if the file is not found

DIR

This command shows the file names that are on the RAMDISK, their lengths, and the amount of available space.

DUMP <addr1,addr2>
or
D <addr1,addr2>

DUMP displays the contents of the logical memory map in hexadecimal and ASCII. If only one address is given, 5 lines of 16 bytes will be shown starting at that address. If two addresses are given, the bytes from <addr1> to <addr2> will be displayed. If no arguments are given, 5 lines of 16 bytes will be displayed starting at the byte after the final byte displayed in the last dump command. Note that on a short line of hex, (less than 16 bytes) the ASCII equivalent characters follow the last hex byte. So in the last line of the first example the 'F' is the ASCII equivalent of the '46' preceding it. Also, for non-printable characters such as carriage return or line feed the ASCII equivalent of '.' is given.

Example 1: (using two addresses)

```
DUMP $8125,$81B5

8125: 76 80 2A FC DE EB 2A 82 80 CD C7 08 69 60 22 F8  v.*...*.....i`".
8135: DE C3 15 81 CD 83 19 01 00 00 C5 C3 4E 81 0A 20  ............N..
8145: 46 49 4E 49 53 48 45 44 2E 21 43 81 C1 CD B3 05  FINISHED.!C.....
8155: C5 C1 CD C6 05 C3 67 3B CD B3 05 C5 C1 CD C6 05  ......g;........
8165: C3 67 3B C5 C1 CD C6 05 C3 67 3B CD 78 06 CD 87  .g;......g;.x...
8175: 2F C3 37 81 21 50 00 CD 78 06 CD 83 19 01 00 00  /.7.!P..x.......
8185: C5 C3 94 81 0A 20 46 49 4E 49 53 48 45 44 2E 21  ..... FINISHED.!
8195: 89 81 C1 CD B3 05 C5 C1 CD C6 05 C3 67 3B 78 06  ............g;x.
81A5: CD 87 2F C3 14 81 C3 67 3B 48 45 20 4B 45 59 20  ../....g;HE KEY
81B5: 46  F
```

Example 2: (using no addresses, it will start at the byte after the last byte displayed and show 5 lines)

```
>DUMP

81B6: 4F 52 20 22 3A 00 EA 00 31 3A 3A 14 B6 20 47 45  OR ":...1::.. GE
81C6: 54 20 4C 45 46 54 4F 56 45 52 20 43 48 41 52 3A  T LEFTOVER CHAR:
81D6: 00 EB 00 00 2D 3A 06 B4 32 30 30 30 3A 00 F0 00  ....-:..2000:...
81E6: 00 2E 3A 1E B8 B1 00 00 3B 22 20 52 49 47 48 54  ..:.....;" RIGHT
81F6: 20 43 55 52 53 4F 52 20 4D 4F 56 45 4D 45 4E 54   CURSOR MOVEMENT
```

Example 3: (using one address)

```
>DUMP $9000

9000: 02 00 02 00 02 00 02 00 02 00 02 00 02 00 02 00  ................
9010: 02 00 02 00 02 00 02 00 02 00 02 00 02 00 02 00  ................
9020: 02 00 02 00 02 00 02 00 02 00 02 00 02 00 02 00  ................
9030: 02 00 02 00 02 00 02 00 02 00 02 00 02 00 02 00  ................
9040: 02 00 02 00 02 00 02 00 02 00 02 00 02 00 02 00  ................
>
```

E <line> or **EDIT <line>**

This command requires an ADM-3A or ADM-5A terminal or a communications program which emulates one of these terminals running on a personal computer. A communications program may be obtained from EMAC which runs on IBM PCs and compatibles. EDIT lets you edit the line number of MTBASIC indicated by <line>. It allows the cursor to be moved to the left and right or to instantly move to the beginning of the line or the end. It also allows characters to be deleted to the left or to the right of the cursor. The EDIT command defaults to the typeover mode which means that whatever is typed replaces what was at the cursor. The insert mode can be enabled which causes all characters to the right of the cursor to be shifted right as you are typing new characters. The EDIT command continues until return is pressed or some undefined character is pressed, in which case the changes to the line are ignored. The default edit keys are:

control-L	(move cursor right)
control-H	(move cursor left)
control-I	(toggle between insert and typeover)
control-D	(delete the character to the right of the cursor)
control-W	(delete the character to the left of the cursor)
control-E	(move cursor to the end of the line)
control-B	(move cursor to the beginning of the line)
return	(enter the edited line to MTBASIC)

The default EDIT control keys can be changed easily by running MTCONFIG.BAS.

END

END is a direct command used only in files containing MTBASIC source programs. During the LOAD command, it indicates to the compiler the end of the file containing MTBASIC statements. Whenever the SAVE command is used to send a program to disk, MTBASIC inserts an END as the last item in the file. The user only needs to use the END command when creating an MTBASIC program off-line with an editor or word processor. In this case, put an END as the last item in the file, without a line number, as follows:

```
      10 PRINT "This program was created using an editor."
      END
```

ERASECHK <addr>,<type>

This command supports the EPROM programmer board (part number E020-08). ERASECHK tells whether the bytes from 0 to <addr> are erased. The value <type> indicates what kind of EPROM you are checking. See the chapter on stand alone applications for more information on making EPROM based MTBASIC programs.

ERROR

ERROR turns on the compiler's runtime error checking software. It is the converse of NOERR. When MTBASIC is first started, all runtime error checking is on. It stays on until explicitly disabled by NOERR. Runtime error checking is essential in most programs to insure that mistakes don't "blow up" the compiler. For example, if a "SUBSCRIPT OUT OF RANGE" error were not detected, a program could write over itself, or even over the compiler, causing unpredictable and undesirable results. Like NOERR, ERROR modifies the code generated during compilation of the program, so it must be specified before the program is RUN or COMPILEd (if you have used NOERR in the MTBASIC session before RUN or COMPILE). ERROR will then remain in effect unless disabled by NOERR.

FORMAT

This command initializes specially allocated memory to be used as a RAMDISK. If you have the RAMDISK option, files are already on it, so make sure you don't need the files because FORMAT make them inaccessible.

GO

The GO command begins execution of the most recently COMPILEd (or RUN) program. If the source code has been modified, or a NEW command has been executed, or there was an error in the last compile, then the NO COMPILED CODE error results. GO will not work for an RCOMPILEd program. GO is typically used to run a program previously compiled via the RUN or COMPILE commands. If the program is very long, then GO is faster than using RUN repeatedly (since RUN must first compile the program again).

IN <port>

This allows you to examine the contents of any input port from 0 to 65535. The byte returned is shown in hexidecimal.

LIST <line 1>,<line 2>

LIST displays the program currently in memory on the system's console. If a PRN command has been issued, the program will also be listed on the printer. If LIST is typed with no arguments, the entire program is listed. If only one line number is specified, only that line, if it exists, will be listed. If two arguments are given, then the program between the two line numbers, inclusive, will be listed.

Examples:

```
     LIST
     LIST 100
     LIST 100,200
```

LISTT <line 1>,<line 2>

This command works the same as LIST only it pauses and prints "More..." at the bottom of the screen and waits for a key to be pressed, before allowing part of the listing to scroll off the screen.

LOAD <filename>

LOAD brings the indicated file into the compiler from RAMDISK. The program is checked for syntax errors as the program is read. The LOAD is terminated when an END command is found in the file. The filename must not be in quotes. For example:

```
LOAD PROG   correct
LOAD "PROG" incorrect
```

If no extension is given, MTBASIC will assume the .BAS extension. This default can be overridden by adding a period after the filename and an extension. LOAD does not erase programs already in memory. The new program will be mixed in with the old, as a function of the line numbers. This provides the ability to merge two or more programs, but if you do not want to merge, use the NEW command before LOADing the new program.

NEW

NEW clears the program currently in memory.

NOERR

NOERR turns off much of the runtime error checking of MTBASIC, resulting in a program that runs much faster. To insure that a program doesn't run wild and destroy itself or the compiler, MTBASIC inserts quite a lot of error checking code into the compiled program. For example, tests are made for "SUBSCRIPT OUT OF RANGE", "NEXT WITHOUT FOR", etc. NOERR also removes the test for a <control-C> . Since <control-C> aborts a running program, a program compiled under NOERR can't be stopped unless the program terminates itself or a non-maskable interrupt (NMI) occurs. Most of these tests are removed by specifying NOERR. Some error checking routines remain, since in some cases the error checking code does not greatly effect execution speed. An increase in execution speed of several hundred percent can often be realized by using NOERR. The compiled program will also be significantly shorter. Be warned, though NOERR permits the user's program to execute erroneously, possibly trashing the MTBASIC operating system. Only fully debugged programs should be compiled under NOERR. When the compiler starts, all error checking is enabled and remains on until NOERR is specified. NOERR affects the way a program is compiled, so it must be specified before the program is COMPILEd or RUN.

OUT <port>,<data>

This instruction allows you to send <data>, which must be less than 256, to the output port <port> which can be any value from 0 to 65535.

PRN

The PRN command causes all output sent to the console to also go to the optional Parallel Printer Card. This includes output generated by the program (via PRINT and FPRINT statements), as well as output during direct commands (such as LIST). The CONSOLE command disables PRN.

RCOMPILE

RCOMPILE is the first command issued when it is desired to make a ROMed version of a program. It causes MTBASIC to convert the program currently in memory to machine language. It differs from COMPILE in that RCOMPILE compiles the program so that it can replace the MTBASIC compiler in memory. If the command "RCOMPILE A" is issued, MTBASIC will attempt to compile the program so it can be used on a board with only 32k of RAM. See the chapter on stand alone applications for more information on making EPROM based MTBASIC programs.

RECEIVE <filename>

This command allows data from the console to be entered directly to the RAMDISK. Using a PC-based communications package such as ECOM or PROCOMM will allow very large programs to be loaded from the PC directly to the RAMDISK. Once the entire program has been RECEIVEd, end the file by sending a control-Z and the command will be terminated. If a control-C is RECEIVEd, the command is terminated without saving the data to RAMDISK. This command also allows you to make a file of direct commands, so that when you LOAD the file, MTBASIC will execute all the commands within the file. For example, if the file contains:

```
NOERR
10 INTEGER X
20 FOR X = 1 TO 200
30 PRINT " This is a BASIC program "
40 NEXT X
RUN
STAT
LIST
NEW
RUN MTDEM.BAS
```

and you type LOAD <filename>, MTBASIC will think the data in this file is being typed at the console. You can only have one direct command that uses the RAMDISK per file, so if a direct command such as DIR is encountered or the program in the file uses the RAMDISK, the LOAD command will be aborted. For this reason it is best to put the command that uses the RAMDISK at the end of the file.

REM and NOREM

These are needed when writing very large applications when your program token space is approaching its limit. After executing the NOREM command, all remarks that are typed at the prompt (or LOADed from RAMDISK) will be affected. Only the line number and REM statement will be copied to the token space. Only the line number and statement will be shown when you list the program. For programs that are to be downloaded from a host to the board you may embed the direct commands in your source (on lines by themselves, with no line numbers) and selectively enable and disable comments. You may also use the commands as direct mode comments, without line numbers and they won't take any token space at all (remember that the command is still executed, though). An example of this follows:

```
10 INTEGER X
20 PRINT "HEY"
NOREM   HERE IS A COMMENT THAT WILL NOT BE SAVED
REM  HERE IS ANOTHER
30  GOTO 20
```

RHEX

The RHEX command sends to the console an Intel hex file which contains all of the runtime routines linked with the program that was compiled using the RCOMPILE command. This command is intended to be used with a communications package such as EMAC's ECOM or PROCOMM to allow the Intel hex file to be loaded into a PC where it can be used to program an EPROM. See the chapter on stand alone applications for more information on making EPROM based MTBASIC programs.

RUN

RUN compiles and executes a program. The program may be stopped by typing control-C (unless the program was compiled after the direct command NOERR) and the line number of the last statement executed will be displayed. The program is not executed if any compilation errors are found.

SAVE <filename>

SAVE stores the program currently resident in compiler memory to RAMDISK. The program is SAVEd in ASCII, so most text editors can modify it. The file name specified must not be in quotes (as with LOAD). SAVE assumes an extension of .BAS if none is given. SAVE always puts an END in the file so the LOAD command will operate properly.

SAVE <filename>.HEX

If a compiled program is in memory, you may save the compiled code (instead the ASCII source code) to the RAMDISK as an Intel hex file. This hex file may be executed using the RUN command, or executed on power up by naming it AUTOSTART.HEX. The purpose of this is to encrypt the program code to provide security in applications that execute from RAMDISK.

STATUS

STATUS displays the start and end address of the token space for the program, of the compiled code, along with the amount of free memory and the starting address of the variables used in the program. The free space may be used for assembly language routines.

EXAMPLE:

```
>LOAD WIND1
>COMPILE

COMPILED
>STAT
```

```
         Start   End     Hex     Dec
         Addr    Addr    Size    Size
Token Space: 8000    855D    055E    1374
Program:     85C9    8E97    08CF    2255
Free Space:  8E98    D50F    4678    18040
Variables:   D510    DFFF    0AF0    2800
User Space:                  0000    0
>
```

TBURN

This command supports the EPROM programmer board (part number E020-08). TBURN first checks to see if the EPROM in the programmer is blank and then it allows you to take a program that was RCOMPILEd and copy it to a 27512 EPROM. See the chapter on stand alone applications for more information on making EPROM based MTBASIC programs.

TYPE <filename>

This command is intended to be used with a communications package (such as ECOM or PROCOMM) to back up RAMDISK files on a personal computer. It sends the data from the file to COM1. It can also be used to view the data in a file. If a control-Z character is encountered, the TYPE command will pause for a moment to allow you to press control-S to stop it from printing to the screen. This is done because, on some terminals, control-Z causes the screen to be cleared.

VERIFY <source>,<dest>,<qty>,<type>

This command supports the EPROM programmer board (part number E020-08). VERIFY tells whether the <qty> number of bytes at <source> match the <qty> number of bytes at <dest>, for the EPROM described by <type>. The command will terminate and give the source address followed by the value at that address, and the destination address followed by the value at that address, if there are differing values. See the chapter on stand alone applications for more information on making EPROM based MTBASIC programs.

An MTBASIC statement is composed of the name of the statement itself (for example, GO TO), along with optional parameters. These parameters may consist of variables, constants, functions, operators, or combinations of all of the above.

VARIABLES

MTBASIC variables are formed by a letter followed by up to six letters or digits. For example, A is a legal variable, as well as A0, A1, HI92, and JUMP. However, 9AB is not a legal variable, nor is A1234567899. String variables are formed the same way, but are followed by a dollar sign. HIYA$, A1$, A9$ are all legal string variables.

A maximum of 255 different variables may exist in any given program. These variables may be in any form within the rules given above. Note that integer or real variables may not have the same name as string variables. For example, the variable A may not be used in the same program that uses the variable A$. MTBASIC will try to use A and A$ as the same variable and a string variable error will result. Similarly, a dimensioned variable must not have the same name as an undimensioned variable (for example, you cannot use B and B(n) in the same program). All variables must be declared at the beginning of the program before any executable code is encountered. In practice, this means that the variable declaration statements (INTEGER, REAL and STRING) should be the first statements in the program. If undeclared variables are found during the compilation, the compiler will display an error message.

For strings, the string length is specified in the STRING statement (for example, B$(80)). If no string length is specified, a string length of 20 will be assigned. The maximum allowable string length is 127. String arrays are defined by specifying the length of each element, followed by the number of elements in the array. STRING A$(10,20) specifies 20 strings, each of length 10. Any element can be accessed just like a singly dimensioned array. A$(3)="123" assigns a value to the third element of the string array. Subscripted variables are specified within the INTEGER and REAL statements. Note that subscripted variables start with the zero dimension, and extend to the maximum dimension specified. Therefore, the statement INTEGER A(10) defines a variable with eleven members, A(0) through A(10).

Variables are not automatically initialized by MTBASIC. This must be done by the program using the LET or CLEAR statements.

NUMBERS

INTEGER variables are stored using a sixteen bit two's complement representation. An integer value can range from +32,767 to -32,768. Positive values which exceed 32,767 will appear as negative numbers.

Real values are four byte (32 bit) IEEE compatible single precision real numbers. This means that approximately 6.5 digits of precision are maintained for real numbers. Many BASIC interpreters and compilers use BCD mathematics or 64 bit representations resulting in high accuracy numbers that require lots of memory. MTBASIC does not support either of these in the interest of maximizing speed. The user must be aware that a real number may not be exactly the number anticipated. For example, since real numbers are constructed by using powers of 2, the value 0.1 cannot be exactly represented. It can be represented very closely (within 2^{-23}), but it will not be exact. Therefore, it is very dangerous to perform a direct equality operation on a real number. The statement IF A= 0.123 (assuming A is real) will only pass the test if the two values are exactly equal, a case which rarely occurs. This is true for all real relational operators, including, for example, the statement IF A>B, if values very close to the condition being measured are being used. Be aware that the number you expect may not be exactly represented by the compiler. If necessary, use a slight tolerance around variables with relational operators.

CONSTANTS

Constants are formed by combining decimal digits with an optional decimal point. Whenever a decimal point is included in a constant, the compiler assumes this constant is a real number. If the constant is expressed without a decimal point, the compiler assumes that the value is an integer. This has significance when combining real and integer values within an expression (see Chapter 7). Even though MTBASIC is smart enough to convert constants between integer and real as required, programs will run more efficiently if modes are not mixed.

MTBASIC also provides a facility for using hexadecimal and binary constants. Hexadecimal constants are specified with a leading dollar sign. So $1AB represents the hexadecimal constant 1AB. Binary constants are specified by a leading percent sign (%) so the number %10110010 represents the binary number 10110010. Decimal points within the binary number can be used as visual markers. For example %1001.1011 and %10.01.1011 both have the binary value of 10011011.

Another type of constant is a System constant. System constants are specified by a leading pound sign (#) and are used the same way as regular constants. The system constants are defined as follows:

#PPORTA	I/O address of PPI port A
#PPORTB	I/O address of PPI port B
#PPORTC	I/O address of PPI port C
#PPICMD	I/O address of PPI control word
#PPORTD	I/O address of control port D
#SPORTD	Address of RAM shadow of data written to PPORTD

For example:

```
      10 PRINT INPORT(#PPORTA) :' PRINT THE DATA INPUT TO PPI PORT A
```

Also there is a constant which contains the execution address of an MTBASIC subroutine.

#FORMAT	Execution address for the format RAMDISK subroutine.

This is intended to be used as follows: `20 CALL #FORMAT` and will operate the same as the FORMAT command (See chapter on Direct Commands).

MODE CONVERSIONS

MTBASIC is unlike many BASICs in that it forces the user to declare the mode of each variable, thereby optimizing the compiler's speed. With all variables predeclared, the compiler is not forced to evaluate all expressions in floating point at run time (which is a very slow procedure), and then convert to integer as the need arises. Instead, the algorithms used in MTBASIC attempt to evaluate all expressions in the output mode (the mode of the variable to which the expression is being assigned).

To make it easier to write programs, MTBASIC provides automatic mixed mode expression evaluation. This means that an expression may consist of a combination of real and integer values. MTBASIC will automatically convert the components of the expression to the proper mode before evaluating it, and will convert the result to the mode of the variable to which the expression is being assigned. This is very convenient for programmers; however, there are some important implications arising from it. Whenever an expression is to be assigned to a real variable, then every component of that expression is evaluated in real mode. Components of the expression which are integer (for example, integer variables), are automatically converted to real before any arithmetic is performed. This conversion takes place entirely within temporary values in the compiler; the integer values themselves are not changed. Whenever a constant is specified with no decimal point, the compiler assumes that it is an integer value. Any constant designated with a decimal point will be assumed to be real. Since the process of converting an integer to a real is relatively slow, faster code will result with real operations when all real operands are specified.

Expressions are defined in terms of parentheses. Whenever an expression in parentheses is encountered, this is treated as a new expression, although it may be part of a larger expression. This has significance when expressions are being evaluated which will be assigned to integer arguments. When the compiler encounters a new expression (one with parenthesis), it attempts to evaluate that expression in the mode of the variable to which it will be assigned. In the case of a real operator this is not important, since all values are converted to real before any operation takes place. With integer variables, however, if any component of an expression is real, the rest of that expression will be converted to real before the operation takes place. A few examples will make this clear.

```
10 INTEGER A
20 A=(1/2)*2
```

In this case, the expression will evaluate to the value zero. All operations specified are integer. Integer operations take place by truncating the result, so 1 divided by 2 evaluates to 0.

```
10 INTEGER A
20 A=(1.0/2)*2
```

This expression also evaluates to zero, but for a different reason. The inner 1.0/2 evaluates to .5, but after the value is calculated, the compiler attempts to convert this back to integer to be in the proper mode for variable A. The integer version of 0.5 is 0.

```
10 REAL A
20 A=(1.0/2)*2.0
```

In this case, the expression will evaluate to 1. Each of the operations is real, so all operations take place in real mode.

Chapter 4: Operators

Operators are connectors within expressions that perform logical or mathematical computations.

These operators work both with integer and real numbers:

+	addition
-	subtraction
*	multiplication
/	division

Some operators are relational. They generate a non-zero result if their condition is met. These operators may be used in mathematical expressions, but they are more frequently used with IF/THEN statements:

>	greater than
<	less than
<> or ><	not equal to
=	relational equality test
>=	greater than or equal (integers and reals)
<=	less than or equal (integers and reals)
AND	logical AND
OR	logical OR

Note that AND and OR are evaluated in integer. Real arguments are converted to integer before AND and OR are evaluated.
All operators are in a hierarchy that defines what operators will be evaluated first. The following is a list, from highest to lowest priority:

*, /
+,-
unary -, >, <, <>, ><
AND, OR

A variable may hold the result of a relational comparison. For example, A = R > 0.

Strings don't support the >= and <= relational operators. Some BASICs use + for string concatenation, but MTBASIC programs must use the CONCAT$ function.

Chapter 5: Statements

A statement is an instruction in a program which specifies what action the program will take. These are the MTBASIC statements:

CALL	Starts an assembly language subroutine
CANCEL	Stops a task
CLEAR	Initializes a group of variables
CLOSE	Closes a file
CLS	Erase the CRT
CODE	Used to insert machine language statements into MTBASIC
CURSOR	Positions the cursor in a window
DATA	Defines a group of constants
DACOUT	Writes a value to the optional D/A channel
DEF	Marks the beginning of a user defined function
DEFMAP	Specifies the section of memory to be used by PEEK, WPEEK, POKE and WPOKE
DELETE	Deletes a RAMDISK file
DEVICE	Selects an I/O device
EEPOKE	Stores a 16 bit word at a specified address in EEPROM
ERASE	Clears the entire CRT
EXIT	Terminates a task
FIELD	Specify record format
FNEND	Marks the end of a user defined function
FOR/NEXT	Loop control
FPRINT	Formatted print
GOSUB	Subroutine call
GO TO	Program branch
IF	Decision
INPUT	Enter data from I/O device
INPUT$	Enter data, including commas
INTEGER	Defines integer variables
INTON	Turns interrupts on
INTOFF	Turns interrupts off
LET	Used for assigning values to variables
NEXT	End of a FOR/NEXT loop
OFF ERROR	Disable error trapping
ON ERROR	Start error trapping
ON GOSUB	Does a GOSUB based on the value of the expression
ON GOTO	Does a GOTO based on the value of the expression
OPEN	Opens a file
OUTPORT	Output to an I/O port
POKE	Modifies a memory location
PPICFG	Configure I/O direction for digital ports on HDR6
PRINT	Outputs data. Also denoted by "?"
PUSH	Push a word on the system stack
PUT	Send a record to a random file
RANDOMIZE	Seeds the random number generator
RDSCLK	Loads an integer array with the real time clock data
READ	Gets data from a DATA statement
REAL	Defines floating point variables
REM	Comment, also denoted by "!" and " ' "
RESTORE	Causes next READ to read the first DATA element
RETURN	Return from a subroutine
RGET	Get a record from a random file
SEEK	Select a record to write or read
SETBAUD	Set the baud rate for a COM port

SETINTS	Directly set or reset interrupts 0-9
START	Begins executing a task
STOP	Halt a program
STORMASK	Sets the interrupt mask value after executing an EXIT
STRING	Defines string variables
TASK	Defines the start of a task
TRACE ON	Prints line numbers as they are executed
TRACE OFF	Disables TRACE ON
VECTOR	Defines interrupt vector
WAIT	Delays a task's execution
WATCHDOG	Pulses the watchdog timer
WCLEAR	Erases a window
WFRAME	Draws an outline around a window
WINDOW	Defines a window
WPOKE	Store 16 bits at the specified address
WRSCLK	Writes the integer array to the real time clock
WSAVE	Saves the contents of a window
WSELECT	Selects a window
WTIMER	Sets the selected 64180 PRT
WUPDATE	Restores a saved window
:	Separates multiple statements on a line

All statements must be preceded by a line number. This line number determines the position of the statement within the program. For example, if a statement is entered with a line number of 10, and later on a statement is entered with a line number of 1, the statement numbered 1 will appear in the program before the statement numbered 10. This is MTBASIC's primary editing facility. To delete a statement, simply type its line number followed by a return. To replace a statement, type the statement's line number followed by the new statement. These procedures are the same as those used on many interpreters. Multiple statements may be on the same line if each statement is separated by a colon. For example:

```
10 PRINT "HELLO "; : PRINT "BOB" : ! OUTPUT IS HELLO, BOB
```

CALL <address>,Arg1, Arg2 ...

The CALL statement begins execution of a machine language subroutine starting at <address>. Execution of the assembly language subroutine continues until a RET instruction is encountered. As with an assembly language CALL, the return address is put on the stack, so a RET will return to the calling MTBASIC program. No registers need be preserved.

The assembly language routine being called may be POKEd into memory by the calling MTBASIC program. To determine a "safe" area of memory for an assembly routine, use the STATUS command to find out where the MTBASIC program ends. The assembly routine may be POKEd between the end of the program and the start of the variable storage. Alternatively, an array can be defined and the program POKEd into the array. The address of the array can be determined using the ADR function, as in the following example:

```
10 INTEGER B(100),J,K
20 DATA $21,$05,$00,$C9 : REM This is Z-80 code.
40 FOR J=0 TO 3
50 READ K
60 POKE ADR(B(0))+J,K
70 NEXT J
80 CALL ADR(B(0))
```

If optional arguments are given after the address of the routine being called, then the address of these arguments are passed. This allows any value to be passed to an assembly routine. Since the addresses of the values are passed, the assembly routine can alter the values before returning to the calling program. Only variables should have their values altered. For example,

```
10 CALL $A000,VAR1,VAR2
```

will compile a call as follows:

```
                    CALL $A000
                    .BYTE <mode of VAR1>
                    .WORD <address of VAR1>
                    .BYTE <mode of VAR2>
                    .WORD <address of VAR2>
                    .BYTE 0 ; signals end of list
```

Modes are assigned as follows:

1 - Integer
2 - Real
3 - String

If arguments are passed, be sure to return to a point after the final 0 in the argument list!

The arguments of the CALL statement can be arrays, constants or variables, however, since MTBASIC references array elements indirectly, data can be passed FROM an array TO a machine language program, but not the other way. If you need to pass an argument back to the MTBASIC program from a machine language routine, do it through an intermediate variable, as follows:

```
50 CALL $9000, K
60 A(3)=K
```

CANCEL <task>

CANCEL stops the specified task from being started again when the schedule interval given in the START statement elapses. CANCEL does not abort the task - only EXIT or STOP can stop the execution of a task. When a START statement is entered, a schedule interval is specified. CANCEL causes the scheduling of the task to cease. When the CANCEL is executed, the task continues executing normally until it EXITs. The task will not run again until it is specifically commanded to via another START statement. CANCEL is useful when a task need only be run once. The task CANCELs itself, as in the example below (see the chapter on multitasking for more information).

```
10 INTEGER I
20 START 1,10
30 PRINT "Enter a number"
40 INPUT I
50 PRINT I
60 GO TO 40
70 TASK 1
80 PRINT "Page Heading" ! Print it only once.
90 CANCEL 1
100 EXIT
```

CLEAR

This will clear all variables that were declared prior to this statement. All variables declared after it remain uninitialized. To demonstrate this, RUN the following example once (to load X and Y) and RUN it again. You will see that X is CLEARed since it is declared prior to the CLEAR statement and Y=5678, which is the value it had after the program ran the first time. If you delete line 20 and RUN the program again, X will be 1234 and Y will be 5678 which are the values that were loaded when it was last RUN.

```
10 INTEGER X
20 CLEAR
30 INTEGER Y
40 PRINT X,Y
50 X=1234
60 Y=5678
```

CLOSE <file number>

The CLOSE statement is used to disassociate a file number from a file name (this association was made with an OPEN statement). CLOSE is required when using write files, so that the file length can be added to the RAMDISK. Errors, if any, are returned by the ERR function.

Example:

```
10 STRING B$
20 FIELD 15,B$
30 B$="MTBASIC is cool"
40 OPEN 1,1,"FILE.DAT"
50 PUT
60 CLOSE 1: ! THIS STORES THE FILE LENGTH TO RAMDISK
70 B$="SOMETHING ELSE":! B$ IS NOT WHAT IT WAS BEFORE
80 OPEN 0,1,"FILE.DAT"
90 RGET :! GET WHAT WAS STORED TO DISK
100 CLOSE 1
110 PRINT B$ :! PRINT THE MESSAGE
```

CLS

CLS is the same as ERASE. It erases the contents of the CRT. MTBASIC must be configured before using CLS (see the chapter on windowing).

CODE <list of hex machine codes>

The CODE statement is used to insert machine language statements in-line with an MTBASIC program. This should ONLY be done by experienced machine language programmers because it is possible that an error with CODE could cause irreparable damage. The arguments given to a CODE statement must be numbers. Expressions like $41-$22 are not allowed. MTBASIC simply copies the argument to memory. The machine code is generated at the indicated line number. You can "flow into" the machine code, or reference it by a GOTO the line number. CODE statements can be put into user defined functions. It is also possible to write a subroutine with CODE statements, and to then access it with a GOSUB; however, you MUST return from the subroutine with an MTBASIC RETURN, as follows:

```
      110 CODE $21,0,0
      120 CODE $3E,0
      130 RETURN
```

You can use any Z-80 register in your code; MTBASIC does not expect you to preserve the registers. You CANNOT use the alternate register set.

CURSOR <row coordinate>, <column coordinate>

CURSOR is used to position the cursor within a window on the screen. Like all window related commands, it will function properly only if MTBASIC has been configured for the terminal or communication package being used.

CURSOR's arguments are the logical row and column coordinates of the desired window position. These are logical values, meaning that they are referenced to the current window's upper left hand corner. CURSOR 0,0, therefore, positions the cursor to the current window's upper left hand corner. This gives the programmer independence from the exact screen layout. The windows can be repositioned at any time by simply changing the WINDOW statement itself. All cursor positioning will remain the same. For example:

```
10 WSELECT 1
20 WINDOW 10,10,20,20
30 CURSOR 10,10
40 GO TO 40
```

Line 30 moves the cursor to the lower right hand corner of the window, since the window is 11 characters on a side.

DACOUT <expr1>,<expr2>

DACOUT sends the value of <expr2>, which must range from 0 to 4095, to the digital to analog convertor channel (0 to 3) selected by <expr1>. This statement requires the digital to analog convertor option (part number E311-04).

Example:

```
10 REM THIS PROGRAM PRODUCES A REPETITIVE ASCENDING RAMP SIGNAL ON
20 REM CHANNEL 0 WHILE PRODUCING A DESCENDING RAMP SIGNAL ON CHANNEL 1.
40 INTEGER X
50 FOR X=0 TO 4095
80 DACOUT 0,X
90 DACOUT 1,4095-X
100 NEXT X
110 GOTO 50 : REM PRODUCE THE NEXT RAMP
```

DATA <data list>

DATA statements are used to define a block of constants which will be read by READ statements. ALL DATA statements must be in your program before the first READ statement. When the first READ statement is encountered, MTBASIC figures out where all of the DATA statements are. For example:

```
10 INTEGER A,D
20 REAL B
30 STRING C$
40 DATA 2,3*3.14, "Howdy, Ma'am"
50 DATA 4
60 READ A,B,C$,D
```

At the end of this program, A=2, B=9.42, C$="Howdy, Ma'am", and D=4.

DEF <function name> {<arguments>}

The DEF statement marks the beginning of a user defined function. The function name must follow the rules for variable names. arguments for the function are optional. Also see FNEND and Chapter 6 for more information.

DEFMAP <address>

The DEFMAP statement is used to provide access to memory outside of the current logical address space. It causes the PEEK and WPEEK functions and POKE and WPOKE statements to operate on memory anywhere in the physical address space of the computer. When MTBASIC is started, DEFMAP defaults to a -1 which has the effect of accessing only the current logical map.

If a DEFMAP is issued with an argument other that -1, all PEEK, POKE WPEEK and WPOKE statements form their physical addresses as follows: **Physical address = (DEFMAP address) * $1000 + (PEEK address) AND $0FFF**

If the PEEK address was $35AC and the DEFMAP address was $82, the physical address would be calculated as follows:

```
       $05AC      PEEK address (the AND $0FFF removes the 3)
      $82000      DEFMAP address * $1000
      $825AC      Resulting physical address
```

Example:

```
10 DEFMAP $12
20 PRINT PEEK ($345): ! The effective address is $12345
30 PRINT PEEK ($FABC): ! The effective address is $12ABC since "F" is removed
```

Note that these addresses are physical addresses - they reference absolute locations in memory. If DEFMAP address -1 is used then all addresses are logical. The full 16 bits of the address are used to access the default memory map set up by MTBASIC. If a PEEK, WPEEK, POKE or WPOKE are used in more than one task, DEFMAP shouldn't be used unless its value remains the same in all the tasks.

DELETE <file name>

This statement deletes the file specified by <file name> from the RAMDISK. Errors, if any, are returned by the ERR and ERR$ functions. For example:

```
10 STRING FILE$
20 PRINT "Enter the name of the file to be deleted"
30 INPUT  FILE$
40 DELETE FILE$
50 IF ERR = 0 THEN GOTO 100
60 PRINT ERR$
70 GOTO 20
100 PRINT FILE$;" has been deleted"
```

DEVICE <device number>

DEVICE causes all subsequent I/O from PRINT, FPRINT, INPUT, and INPUT$ statements to be sent to the device specified by <device number>. Errors, if any, are returned by the ERR and ERR$ functions. The file number definitions are:

-1 = Parallel Printer Interface board
0 = COM1 (console)
1 = RAMDISK file 1 (The file number is assigned to a file name by an OPEN statement)
2 = RAMDISK file 2
3 = RAMDISK file 3
4 = RAMDISK file 4
5 = COM0
6 = COM1
7 = COM2. This optional serial port requires that CSI/O and INT2 interrupts be enabled. It has a 16 character receive buffer.
8 = COM0. This has an interrupt driven circular input buffer with X-ON/OFF protocol. COM0 RX interrupts must be enabled (see SETINTS) before selecting this device. If an X-OFF (11h) is received, no characters will be transmitted till an X-ON (13h) is received. If the input buffer is about full an X-OFF will be transmitted from COM0 and when it's about empty an X-ON will be transmitted.
9 = When this device is used for input, the data comes from the optional keypad interface (see table 1 for definition). CSI/O and INT2 interrupts must not be disabled (they are enabled by default). When this device is used for output, the data goes to the on-board LCD interface. Note that carriage return and line feed characters are not translated so all PRINTs should end with a semicolon to suppress these. This device accepts the codes shown in table 2.

Table 1: Characters returned from a 4x4 keypad

```
ABCD
EFGH
IJKL
MNOP
```

Table 2:

LCD code	**function**
16	Initialize LCD display
20	clear the display
21	move cursor to line 1
22	move cursor to line 2

A code can be sent to the display by first selecting DEVICE 9 and then using the line: `"PRINT CHR$(`***<code>***`);"`
14= LCD-KEYPAD board option

Note: Accessing devices -1, 7, 9 or 14 without having the appropriate hardware options installed could crash your program.

The DEVICE statement sets the device number for the task that issues the statement. Each task may have a different active device number (see example below)

```
10 INTEGER I
20 I=0
30 PRINT "This is the beginning."
40 START 1,1000
50 START 2,200
70 IF I=10 THEN STOP
80 GO TO 40
90 TASK 1
100 DEVICE 0
```

...continued on next page

```
110 PRINT "MESSAGE TO COM1"
120 EXIT
130 TASK 2
140 DEVICE 5
150 PRINT " MESSAGE TO COM0"
160 I=I+1
200 EXIT
```

EEPOKE <expr1>,<expr2>

EEPOKE places the 16 bit value of <expr2> into the EEPROM at the address specified in <expr1>. Valid values for <expr1> are 35-63, so a subscript out of range error will be displayed if <expr1> is greater than 63 or less than 35. You can access the addresses below 35 if you add $0B00 (2816 in decimal) to the address desired. Probably the only addresses you would want to access would be 1 and 2 which are the addresses of the baud rates for COM1 and COM0 respectively. Be careful when changing COM1 because if you change the baud rate to a value that your terminal doesn't support, you will have severed the communication between the terminal and the board and you will not be able to change the baud back!

WARNING, you can write to EEPROM no more than 10,000 times, so this command should be used sparingly.

Example:

```
10 REM THIS PROGRAM CHANGES THE BAUD RATE OF COM0
20 INTEGER BAUD,UARTADDR
30 REM THE ADDRESS IN THE EEPROM OF THE BAUD FOR COM0
31 UARTADDR = 2 + 2816
40 PRINT " ENTER THE BAUD FOR COM0"
50 INPUT BAUD
60 EEPOKE (UARTADDR),BAUD
```

ERASE

ERASE clears the contents of the terminal screen. MTBASIC must be configured before using this statement.

EXIT

EXIT causes the currently executing task to abort. When the EXIT is encountered, the task stops until the schedule interval specified in the START statement elapses, at which point the task starts over again from its beginning. Note that EXIT does not stop the task from being rescheduled; only CANCEL can do that. Whenever EXIT is executed, it automatically turns interrupts back on (i.e., simulates an INTON). This feature is needed by hardware device interrupt handlers to insure that the device interrupt handler can return just as interrupts are re-enabled. The following program prints a message 10 times. EXIT causes the task to stop, but allows it to be rescheduled again. The CANCEL causes final, and complete, termination of the task (for more information see the chapter on multitasking).

```
20 INTEGER I
30 I=0
40 START 1,100
50 IF I<> 10 THEN 50
60 CANCEL 1
70 STOP
80 TASK 1
90 PRINT "The horse to bet on in the fifth is ..."
100 I=I+1
110 EXIT
```

FIELD <length, variable, length, variable...>

The FIELD statement defines the contents of a record for random file I/O. FIELD is used in conjunction with PUT and RGET for doing the

I/O. Chapter 10 gives an example of the use of FIELD. FIELD must be used with pairs of arguments. The first element of a pair is the number of bytes to be transferred for the second element of the pair, which must be a variable. Unlike most BASICs, MTBASIC also allows you to use REALs and INTEGERs within a field statement, although arrays cannot be used. INTEGERS are of length 2 and REALs are of length 4.
For example,

```
10 REAL Y
20 INTEGER X
30 STRING A$,B$
40 FIELD 2,A$,2,X,3,B$,4,Y
```

specifies that 2 bytes are to be transferred for A$, 2 for X, 3 for B$, and 4 for Y.

FNEND

The FNEND statement marks the end of a user defined function. Also see DEF and Chapter 6.

FOR <variable> = <expr 1> TO <expr 2> {STEP <expr 3>}

This statement will start execution of all statements between itself and the first NEXT it finds. Initially, <variable> is set to <expr 1>. During each iteration of the loop, <variable> (which must be a nonsubscripted variable) is incremented by 1 (default value) or by <expr 3>. (The difference between <expr 1> and <expr 2> must not be more than 32768.) The statements within the loop will be executed until <variable> is equal to or greater than <expr 2>. Loops may be nested like this:

```
10 INTEGER K,J
20 FOR K=1 TO 10
30 FOR J=1 TO 10
40 PRINT K,J
50 NEXT J
60 NEXT K
70 STOP
```

FPRINT <format>, <expr list>

FPRINT allows formatted printing on the screen. Using the <format> string (explained below), the list of expressions <expr list> is printed. FPRINT is a much more sophisticated version of the PRINT statement, since FPRINT allows the programmer to precisely control the format of data output by the program. If the width specified for a field is not wide enough to contain the item printed, then asterisks are printed. For example, if a format of H2 is specified, and the number output is $3344, then ** will be printed. Whenever a field overflow of this sort occurs, the field width specified in the format statement (in this case 2) will be filled with asterisks. In the <format> string the following symbols are valid:

Hn	Prints **n** hexadecimal digits, truncating if necessary. Leading zeros are printed.
In	Prints **n** integer digits. Leading zeros are converted to blanks before being printed.
Qn	Prints **n** integer digits. Leading zeroes are printed (negative numbers should not be used).
Sn	Prints **n** characters of a string.
Xn	Prints **n** spaces
Fm.n	Prints m digits before the decimal point and n digits following it. Leading zeros are converted to spaces, and trailing zeros are left as zeros. No more than 6 positions after the decimal point are allowed.
Rm.n	Prints **m** digits before the decimal and **n** digits after. Leading and trailing zeroes are printed (negative numbers should not be used).

Z Suppresses the carriage return at the end of the line. This is only legal at the end of the format string.

Example 1: X = $2AB and Y = 123.123456

```
10 FPRINT "H3,X1,F3.4",X,Y
```

This will produce the following output:

```
2AB 123.1234
```

Example 2: A$ = "Hello" and B$ = "John Doe"

```
20 FPRINT "S5,X1,S4",A$,B$
```

This will produce the following output:

```
Hello John
```

Example 3: A$ = "I1,X5,I1", B = 3 and C = 9

```
30 FPRINT A$,B,C
```

This will produce the following output (3 and 9 are separated by 5 spaces) :

```
3     9
```

GOSUB <line number>

This is MTBASIC's subroutine call technique. GOSUB is a shortened form of GO to SUBroutine. When a GOSUB is encountered, program execution continues at <line number> until a RETURN is executed. The following program will calculate (J*4)*(J-2) for the numbers 1 through 5 and for 20 through 30. In this program a GOSUB is used instead of typing the formula twice.

```
10 INTEGER J,K
20 FOR J=1 TO 5
30 GOSUB 120
40 PRINT "K="; K
50 NEXT J
60 FOR J=20 TO 30
70 GOSUB 120
80 PRINT "K="; K
90 NEXT J
100 STOP
110 REM THIS IS THE SUBROUTINE
120 K=(J*4)*(J-2)
130 RETURN
```

GOTO <line number>

GOTO transfers control to the line number given as its argument. For example:

```
10 GOTO 30
20 PRINT " THIS WON'T BE PRINTED "
30 GOTO 10
```

IF <rexpr> THEN <statement> or <line number>

This statement allows branches to <line number> if the relational expression <rexpr> evaluates to be true. If a statement follows the THEN, it will be executed if <rexpr> is true. If <rexpr> evaluates to be false, program execution continues at the next line in the program. For integers and real numbers, the relational operators are: =,<>,><,<,>, AND, and OR. For strings, only =, <>, and >< may be used. Note that the IF statement actually tests the result of an expression. Transfer to the indicated line number takes place if the expression is non-zero, therefore, statements of the form IF A THEN 100 are legal.

Example:

```
10 STRING A$
20 PRINT "Is it sunny? ";
30 INPUT A$
40 IF A$="YES" THEN 70
50 IF A$="NO" THEN PRINT "Get an umbrella": STOP
60 PRINT "Get a suntan"
70 STOP
```

INITPRN

This command initializes the printer that is connected to the optional Parallel Printer Adapter. Avoid using this command if anything other than the Parallel Printer Adapter is connected to HDR2. This command is only necessary once in a program, and after that you can direct output to the printer by using the command, DEVICE -1. The two errors that can be produced by this command are "No Paper" and "Printer Not Ready" and they must be handled via the ONERROR statement. You can find the error number through the ERR function and the error message through the ERR$ function.

Example:

```
10 ONERROR 130
20 INTEGER X
30 PRINT "INITIALIZING PRINTER!"
40 INITPRN
50 DEVICE -1
60 PRINT "PRINTER TEST"
70 DEVICE 0
80 PRINT "CONSOLE TEST"
90 DEVICE -1
100 PRINT "PRINTER TEST 2"
110 STOP
120 REM ERROR HANDLING ROUTINE
130 PRINT ERR$
140 PRINT "ERROR #";ERR
150 PRINT "PRESS A KEY TO TRY AGAIN"
160 X = GET
170 ERASE
180 GOTO 30
```

INPUT <var> {,<var>...}

The INPUT statement stops the program until the user types a value for the variable <var> and ends it with a carriage return. If <var> is an integer or a real, an error message will result if a string is typed and then the request for input will be made again.
Multitasking operations are not affected by INPUT. An INPUT statement stops only the task that issues the INPUT, not any other task.

INPUT statements cannot read commas when string variables are given. The commas delimit input items to the program. By the same token, control characters cannot be read by INPUT. INPUT$ will, however, read commas in string variables.

The following program asks the user what his name is and then greets him by name.

```
10 STRING A$
20 PRINT "What is your name?"
30 INPUT A$
40 PRINT "Hello, ";A$
50 STOP
```

This program requests the user's age and then tells how many days he has lived.

```
10 INTEGER K
20 PRINT "How old are you?"
30 INPUT K
```

```
40 PRINT "You have lived ";
50 PRINT 365*K;" days"
```

The following program reads from one task, and encourages the user to type faster from the other task. Task 1 runs during the INPUT.

```
20 INTEGER K
30 START 1,200
40 PRINT "Enter a number from 1 to 10"
50 INPUT K
60 STOP
70 TASK 1
80 PRINT "Come on, hurry up!"
90 EXIT
```

INPUT$ <var> {,<var>}

INPUT$ is identical to INPUT, with one exception. If a string variable is specified, the INPUT$ will read whatever is entered, including commas and most control characters.

INTEGER <var> {,<var>......}

INTEGER allows the user to specify that a variable or group of variables is to be handled as a 16 bit two's complement integer (useful range is -32,768 to 32,767). MTBASIC requires that all variables be declared as INTEGER, REAL, or STRING.

The INTEGER statement must occur before any executable statement in the program. Generally, all INTEGER, REAL, and STRING statements are the first statements in a program. Integer arrays are also specified using this statement. In the case of an array, the variable name must be followed by the maximum dimensions expected. Arrays may not include more than 2 dimensions. Array dimension 0 is always present. If an array is dimensioned as A(10), then A(0) through A(10) may be referenced.

Example:

```
10 INTEGER K,J(10),T(200,10)
20 INTEGER A
```

INTON and INTOFF

These statements enable and disable software and hardware interrupts. INTOFF is used to keep the lead task from being interrupted and INTON allows the lead task to be interrupted. To enable and disable interrupts within a task which is referred to by a VECTOR command, use the SETINTS, and STORMASK statements. Refer to the chapter on interrupts for more information. The program below, will toggle the interrupts off and on with each key pressed.

Example:

```
10 INTEGER TOGGLE,DUMMY
20 START 1,50
30 START 2,75
40 TOGGLE = 1
50 DUMMY=GET : REM THIS IS USED FOR A PAUSE ONLY
60 IF TOGGLE THEN PRINT "INTOFF" : TOGGLE = 0 : INTOFF : GOTO 50
70 IF TOGGLE = 0 THEN PRINT "INTON": INTON: TOGGLE = 1
80 GOTO 50
100 TASK 1
110 PRINT "TASK 1"
120 EXIT
200 TASK 2
210 PRINT "TASK 2"
220 EXIT
```

NEXT <variable>

The NEXT statement terminates FOR loops. A NEXT must appear for every FOR.

OFF ERROR

OFF ERROR cancels an ON ERROR command. It takes no arguments.

EXAMPLE:

```
10 ON ERROR 1000
20 OFF ERROR
30 STOP
```

ON ERROR <line number>

When MTBASIC encounters a RAMDISK error or printer error (when using the optional Parallel Printer Adapter) it will display an error message and stop the program. The ON ERROR statement can be used to transfer control to another part of your program when an error is found. The line number specified in the ON ERROR will receive control when an error is detected. The following program will print an error message if a problem is found during the OPEN statement.

```
10 ON ERROR 100
20 OPEN 0,1,"FILE"
30 STOP
100 PRINT ERR$
```

ON <expr>, GOSUB line#, line#, ...

ON GOSUB is a computed transfer statement. It causes the program to branch, via GOSUB , to a line number based on the value of the <expr>. The expression must evaluate to a number from 1 to the number of line numbers given in the command. If the expression evaluates to 1, then the program branches to the first line number. If it is 2, then the program branches to the second line number, and so on. If the expression evaluates to a number larger or smaller than the number of line numbers given, then the error message "Line number does not exist" will appear.

```
10 INTEGER I
20 INPUT I
30 ON (I+2), GOSUB 50, 60, 70, 80, 90
40 STOP
50 PRINT 50
55 RETURN
60 PRINT 60
65 RETURN
70 PRINT 70
75 RETURN
80 PRINT 80
85 RETURN
90 PRINT 90
95 RETURN
```

ON <expr>, GOTO line#, line#, ...

ON GOTO is a computed transfer statement. It causes the program to branch, via GOTO , to a line number based on the value of the <expr>. The expression must evaluate to a number from 1 to the number of line numbers given in the command. If the expression evaluates to 1, then the program branches to the first line number. If it is 2, then the program branches to the second line number, and so on. If the expression evaluates to a number larger or smaller than the number of line numbers given, then the error message "Line number does not exist" will appear.

EXAMPLE:

```
10 INTEGER I
20 INPUT I
```

```
30 ON (I+2), GOSUB 50, 60
40 STOP
50 PRINT 50
55 STOP
60 PRINT 60
```

OPEN <flag>, <device number>, <file name>

The OPEN statement is used to assign a device number to a RAMDISK file, and to prepare that file for I/O operations. An OPEN is required before any file is used. Errors, if any, are returned by the ERR and ERR$ functions.

The value for <Flag> is either a 0 or 1. 0 indicates that the file will be used for read operations. 1 indicates that the file will be used for write operations. <Device number> can only be 1 in the current version of MTBASIC. This option is left in for compatibility with future versions of MTBASIC which may allow a number from 1 to 3 in which up to 3 files may be open at any one time. <File name> is an ASCII string containing the file name. The filename will be shown in the directory with all characters converted to upper case and space characters converted to underscores ("_"). The following are examples of valid OPEN statements:

```
10 OPEN 0,1,"REC.DAT" : REM opens REC.DAT for READ operations
20 CLOSE 1
20 A$="STUFF"
30 OPEN 1,1,A$ : REM opens STUFF for WRITE operations
```

OUTPORT <expr 1>, <expr 2>

The OUTPORT statement sends the byte <expr 2> to output port <expr1>. No check is made to see if anything is connected to the port. OUTPORT sends only the lower 8 bits of <expr 2> to the port. The following program sends the numbers FF hex to 0 to the high current driver port. This port is accessible through the odd numbered pins 1-15 of HDR6.

```
1  INTEGER X
5  PPICFG %1011 ' SET UP PORTB FOR OUTPUT, THE REST ARE INPUTS
10  FOR X=$FF TO 0 STEP  - 1
20  OUTPORT (#PPORTB),X
30  WAIT 1
40  NEXT X
```

POKE <expr 1>, <expr 2>

POKE places the value of <expr 2>, which must be from 0 to 255, into memory location <expr 1>. The following program places a C9H at 8700H.

```
10 POKE $8700,$C9
```

PPICFG <expr>

This configures the directions for PPI ports A, B and the lower and upper 4 bits of port C based on the value of <expr> which is formatted as follows:

bit	
0	port C upper 4 bits (1 = input, 0 = output)
1	port C lower 4 bits (1 = input, 0 = output)
2	port B (1 = input, 0 = output)
3	port A (1 = input, 0 = output)

The program below makes the upper half of port C into inputs and the lower half of the port C and port B into outputs (remember that if you want to detect the level of the port B output using a scope or meter, pullup resisters should be used) . If the binary value that is applied to the upper half of port C is less than 12 then the OUTPUT BIT (see diagram below) corresponding to the input value will be set to 1, with all other OUTPUT BITs reset to 0. For example, if the binary value being input to the upper half of port C is 8 then OUTPUT BIT 8 will be 1. This program simulates a 4 to 12 decoder.

```
    PORT C (lower)         PORT B
```

```
 3  2  1  0     7  6  5  4  3  2  1  0   PORT BIT NUMBER

11 10  9  8     7  6  5  4  3  2  1  0   OUTPUT BIT NUMBER
```

EXAMPLE

```
      10 INTEGER DATIN
      20 PPICFG %0001 : REM UPPER OF PORT C IS INPUT, THE REST ARE OUTPUTS
      30 DATIN = INPORT(#PPORTC): REM Get data from the upper 4 bits of port C
      40 DATIN = DATIN / 16: REM Move data from the upper 4 bits to the lower
      50 DATIN = EXP(LOG(2)*DATIN): REM Two to the power of DATIN
      60 OUTPORT(#PPORTB),DATIN: REM OUTPORT always sends only the lower 8 bits
      70 REM Output the upper 8 bits to port C.  Since only the lower 4 bits of
      80 REM  port C are output, output will only go to the lower 4 bits
      90 OUTPORT(#PPORTC),DATIN / $FF
      100 GOTO 50
```

PRINT <expr>, {<expr>...}

PRINT sends output to the currently active device, based on the setting of the last DEVICE statement. The console is the default device. Commas or semicolons may separate print items. Commas cause each item to be separated into columns, while semicolons cause the items to be run together. If the last thing on the PRINT line is a semicolon, no carriage return or line feed will be printed. A question mark is equivalent to PRINT.

```
PROGRAM                                  OUTPUT
10 STRING N$
20 PRINT "Bagels"                        Bagels
30 PRINT 1;2                             12
40 N$="Alice"
50 PRINT N$; " is cool"                  Alice is cool
60 ? "where's the print"                 where's the print
```

Note that program lines 10 and 40 do not produce an output as the OUTPUT column would seem to indicate.

PUSH <expr>

This statement is used to put <expr> on the system stack. The most common use is to pass data to machine code that is embedded using the CODE statement. Each PUSH that is before the CODE statement(s) should have its corresponding POP in the machine code. Only integers may be passed directly, but you may pass the addresses of REALs, STRINGs or arrays using the ADR(<var>) function. The following example PUSHes the address of X, and the value of Y and the machine code adds Y to X and stores the value in X.

```
10 INTEGER X,Y
20 X=4 : Y=10
30 PUSH ADR(X)
40 PUSH Y
50 CODE $C1,$E1,$5E,$23,$56 ' POP BC, POP HL, LD E,(HL), INC HL, LD D,(HL)
60 CODE $EB,$09,$EB,$72,$2B,$73 ' EX DE,HL, ADD HL,BC, EX DE,HL, LD (HL),D, DEC HL, LD(HL),E
70 PRINT X
```

PUT

PUT is used in conjunction with FIELD to perform output to an OPENed RAMDISK file. FIELD defines the format of a record; PUT writes a record to RAMDISK. PUT takes no arguments. Every time PUT is executed, the next record is written to the RAMDISK. When PUT is executed, the current value of the variables defined in the FIELD are written to disk. See the chapter on file and device I/O for more information.

RANDOMIZE

The RANDOMIZE statement reseeds the random number generator. To make sure the numbers generated by RND are truly pseudo-random, it is best to use this statement before RND is used. See RND in the chapter about functions for an example.

RDSCLK <9 element integer array>

This command supports the optional real time clock (part # E010-3). RDSCLK loads the 9 element integer array in the following format:

element	description of data	range
0	hundredths of seconds	0 - 99
1	seconds	0 - 59
2	minutes	0 - 59
3	hours	1 - 12 (12 hr mode), 0 - 23 (24 hr mode)
4	day	1 - 7
5	date	1 - 31
6	month	1 - 12
7	year	0 - 99
8	mode	0 - 2 (0=pm, 1=am, 2=24 hr mode)

Example:

```
10 INTEGER CLOCK(8),X
11 STRING I$(6)
15 ERASE
20 RDSCLK CLOCK(0)
25 CURSOR 0,0
30 PRINT "HUNDREDS ";CLOCK(0);
40 PRINT "SECONDS ";CLOCK(1);
50 PRINT "MINUTES ";CLOCK(2);
60 PRINT "HOURS ";CLOCK(3);
70 PRINT "DAY ";CLOCK(4);
80 PRINT "DATE ";CLOCK(5);
90 PRINT "MONTH ";CLOCK(6);
100 PRINT "YEAR ";CLOCK(7);
110 PRINT "MODE ";
120 IF CLOCK(8)= 0 THEN PRINT " PM"
130 IF CLOCK(8)= 1 THEN PRINT " AM"
140 IF CLOCK(8)= 2 THEN PRINT " 24H"
150 STOP
```

READ <variable list>

READ loads the variables in its argument list with values from a DATA statement. As successive READs are executed, data is taken from each DATA statement in the program. For example:

```
10 INTEGER I,J,K
20 DATA 1,2
30 DATA 4
40 READ I,J,K
```

After line 40, I=1, J=2, and K=4. Note that all DATA statements must appear before the first READ in the program. If another READ is encountered at this time, MTBASIC will start reading at the start of the first data statement.

REAL <var> {,<var>...}

REAL allows the user to specify that a variable or group of variables is to be handled as a 4 byte floating point number (one with a decimal point). MTBASIC requires that all variables be declared as INTEGER, REAL or STRING. The REAL statement must occur before any executable statement in the program. Generally, all INTEGER, REAL, and STRING statements are the first statements in a program.

REAL arrays may also be specified using this statement. For arrays, the variable name must be followed by the maximum dimensions expected. Arrays may not include more than 2 dimensions. Array dimension 0 is always present. If an array is dimensioned as A(10), then A(0) through A(10) may be referenced. For example:

```
10 REAL S,T(10),G(2,5)
```

REM <text>   or ! <text>   or ' <text>

REM or ! or ' precede comments. The ! or ' can be anywhere on a line, without using a colon as a statement separator. Nothing is done with comments during compilation. They are for the user's benefit.

RESTORE

This statement causes the next READ statement to start reading at the first element of the first DATA statement in the program. The program below will READ elements from the DATA statement and display them.

Example:

```
      10 INTEGER X$,Y$,Z$
      20 DATA "FIRST","SECOND","THIRD","A","B","C","1","2","3"
      30 READ X$,Y$,Z$
      40 PRINT X$,Y$,Z$
      50 READ X$,Y$,Z$
      60 PRINT X$,Y$,Z$
      70 RESTORE
      80 READ X$,Y$,Z$
      90 PRINT X$,Y$,Z$
      100 STOP
```

RETURN

RETURN ends a subroutine started by a GOSUB. After the RETURN, execution of the program continues at the line number following the GOSUB statement.

Example:

```
      10 GOSUB 100
      20 GOSUB 100
      30 STOP
      100 PRINT "SUBROUTINE"
      110 RETURN
```

RGET

RGET is the converse of PUT. It reads a record from a file whose record structure has been defined by FIELD. Whenever RGET is executed, the next record in the file is read into the variables in the FIELD statement. See the chapter on file and device I/O for more information.

SEEK <record>

This statement allows you to select a record to write or read. It assumes a random I/O file has been OPENed and that a record has been defined using the FIELD statement, and that the random I/O file device is selected. SEEK sets the internal record pointer to <record>. If that record has not been written yet, an End of file error will occur (EOF) and the record pointer will point to the file space following the last record in the file (if the file is empty, it will point to the beginning of the file). For example, if you SEEK 32767 and only records 0 to 4 have been written, an EOF will occur and the record pointer will point to the file space following record 4 (for more information see section on File and Device I/O).

Example:

```
10  INTEGER X,Y
20  STRING A$
30  DELETE "TEST.DAT"
40  OPEN 2,1,"test.dat": ' open device 1 for random i/o
50   IF  ERR THEN  PRINT ERR$: STOP : ' CHECK FOR ERR ON OPEN
60  FIELD 2,X,9,A$: ' define record
70  FOR X=0 TO 25
80   A$=CONCAT$("--DATA--",CHR$(65 + X)): ' CHANGE X TO A LETTER
90  PUT : ' STORE THE RECORD
100   IF  ERR THEN  DEVICE 0: PRINT ERR$: STOP : ' CHECK FOR ERR ON PUT
110  PRINT "WRITING ";X;"  ";A$
120  NEXT X
130  ' NOW READ THE RECORDS BACKWARDS
140  FOR Y=25 TO 0 STEP  - 1
150  DEVICE 1
160  SEEK Y
170  RGET
180  DEVICE 0
190  PRINT X,A$
200  NEXT Y
```

SETBAUD <COM#>, <baud>

This allows the baud rate of the communication port <COM#> (which can be 0 or 1) to be changed to the baud rate of <baud>. The allowable baud rates are 300, 600, 1200, 2400, 4800, 9600, 19,200 and 38,400 is available. Do not use any baud rate other than the ones specified. Other values will give unpredictable results.

Example:

```
10 SETBAUD 0,19200:! This is the baud rate for  COM0
20 SETBAUD 1,600:! This is the baud rate for  COM1
```

SETINTS <expr>

SETINTS directly enables or disables interrupts depending on the value of <expr>.

The bit positions and their corresponding interrupts are as follows:

bit #	**name**	**bit#**	**name**
0	int0	7	COM 1 Tx
1	int1	8	COM 1 Rx
2	int2	9	COM 0 Tx
3	timer0	10	COM 0 Rx
4	timer1	(11-14 not used)	
5	dma0	15	CSI/O
6	dma1		

To allow enabling of an interrupt, the corresponding bit position must have a 1 in it. For example, to enable COM1 receiver interrupts the binary value 1.0000.0000 would be used (remember that the period in a binary number is just a visual marker and does not affect the value of the number).

```
        10 SETINTS %1.0000.0000 : REM ENABLE COM1 RX
```

A combination of interrupts can be enabled by allowing 2 or more bits to be set.

```
      10 SETINTS %101.1001: REM ENABLE INT0, TIMER0, TIMER1, DMA1
```

If there is more than one interrupt enabled, and you want to disable one, do another SETINTS statement but just enable the desired interrupts.

```
       10 SETINTS %111 :!ENABLE INT0, INT1, INT2
       20 SETINTS %011 :!ENABLE INT0 & INT1.  INT2 IS DISABLED
```

START <task number>, <schedule interval>

START is used within a multitasking program to start execution of a task. The specified task in <task number> starts execution 1 tic after the START statement is executed. If the specified task EXITs, then it is automatically restarted after the number of tics in <schedule interval> has elapsed. This provides a convenient method of making something happen periodically. The schedule interval must be in the range of 1 to 32,767 (see the chapter on multitasking). For example:

```
20 START 1,100
30 GO TO 30
40 TASK 1
50 PRINT "Task running"
60 EXIT
```

In the above example, task 1 is started by line 20. Since task 1 EXITs (line 60), the schedule interval on line 20 causes task 1 to restart every time 100 tics elapse after task 1 EXITs. In the following example, the schedule interval has no effect, since task 1 never EXITs.

```
20 START 1,20
30 GO TO 30
40 TASK 1
50 PRINT "TASK": GO TO 50
```

STORMASK <expr>

When an interrupt service routine has finished, MTBASIC enables interrupts according to the value of <expr> given in the most recently executed STORMASK statement. See the chapter on interrupts for more information.

The bit positions and their corresponding interrupts are as follows:

bit #	**name**	**bit#**	**name**
0	int0	7	COM 1 Tx
1	int1	8	COM 1 Rx
2	int2	9	COM 0 Tx
3	timer0	10	COM 0 Rx
4	timer1	(11-14 not used)	
5	dma0	15	CSI/O
6	dma1		

As with the SETINTS statement, an interrupt is enabled by putting a 1 in the bit position corresponding to the interrupt. For example, to enable COM1 receiver interrupts the binary value 100000000 would be used.

```
      10 STORMASK %1.0000.0000 : REM ENABLE COM1 Rx
```

A combination of interrupts can be enabled by allowing 2 or more bits to be set.

```
      10 STORMASK %101.1001: REM ENABLE INT0, TIMER0, TIMER1, DMA1
```

STOP

This statement stops execution of the program.

STRING <var> {,<var>...}

STRING allows the user to specify that a variable or group of variables is to be handled as a string. MTBASIC requires that all variables be declared as INTEGER, REAL, or STRING. The STRING statement must occur before any executable statement in the program. Generally, all INTEGER, REAL, and STRING statements are the first statements in a program.

Variables in the STRING statement may have a maximum string length specified by enclosing the maximum length in parenthesis. If no maximum is given, a maximum length of 20 characters is assumed. A maximum length of 127 may be specified.

A string array may be specified by giving two parameters after the string's name. The first is the length of each string element. The second argument is the number of elements in the array. For example:

```
10 STRING A$(10,20)
```

defines a string array A$ containing 20 strings, each of length 10.

TASK <task number>

This statement marks the beginning of a task. All tasks, other than the lead task (main program), must begin with a TASK statement. The TASK statement must be on a line by itself (not on a multi-statement line). The task number is a unique digit from 1 to 9 used to identify the task.

The task numbers in a program must include all numbers between 1 and the highest task used. For example, it is OK to specify tasks 1, 2, and 3, but specifying tasks 2, 4, and 5 will not work; tasks 1 and 3 must be used. If any gaps exist, tasks numbered after the gap will never be executed. In the previous example, tasks 4 and 5 would never be executed. For more information, see the chapter on multitasking.

In the following example, task 1 is executed every 100 tics. Line 40 marks the start of task 1.

```
10 START 1,100
20 PRINT "Main program"
25 WAIT 100
30 GO TO 20
40 TASK 1
50 PRINT "Task 1"
60 EXIT
```

TRACE ON and TRACE OFF

TRACE ON causes the line number of each line to print as it is executed. TRACE OFF disables TRACE ON.
Note that the line numbers will be output to the selected device (as specified by a DEVICE statement), whether it is console, or printer.

Example:

```
10 INTEGER K
20 FOR K=1 TO 20
30 IF K>15 THEN TRACE ON
40 NEXT K
```

VECTOR <interrupt #>, <task #>

VECTOR, is used to link a source of hardware interrupts into MTBASIC. When VECTOR is executed, a jump command will be inserted into a section of RAM which will cause the task to start when the interrupt is received. This is useful for device interrupt handlers.

VECTOR must be issued before interrupts are received. Note that MTBASIC provides no hardware support of an interrupting device. If the device needs resetting, or any other interaction, the INPORT or PEEK functions, or the POKE or OUTPORT statements allow the user to service the device (see the chapter on interrupts). Below are the interrupt numbers followed by their corresponding interrupt.

<int#>	INTERRUPT
0	INT0
1	INT1
2	INT2
3	TIMER 0
4	TIMER 1

5	DMA channel 0 (DMA 0)
6	DMA channel 1 (DMA 1)
7	Clocked Serial I/O port (CSIO)
8	COM 1 (ASCI 0)
9	COM 0 (ASCI 1)

The following program prints "TIC." after 60 interrupts from TIMER1 within the 64180 microprocessor.

```
10  INTEGER X,DUMMY
20  X = 0
30  INTON
40  VECTOR 4,1:! CAUSE TIMER1 INTERRUPTS TO EXECUTE TASK 1
50  SETINTS  16 :! ENABLE TIMER1 INTERRUPTS
60  STORMASK 16 :! RE-ENABLE TIMER1 WHEN EXITING TASK 1
70  WTIMER 1,7,$1388:! ENABLE TIMER1 AND DOWN COUNT (60 Hz)
80  IF X > 59 THEN ?"TIC."; : X = 0
90  GOTO 80: ! LOOP
100 TASK 1 :! THIS TASK INCREMENTS X WITH EACH INTERRUPT
110 DUMMY = INPORT($10) :!THIS STATEMENT AND THE FOLLOWING..
120 DUMMY = INPORT($15) :!.CLEARS THE TIMER1 INTERRUPT FLAG
130 X=X+1
140 EXIT
```

WAIT <expr>

The WAIT statement delays the current task from executing until <expr> tics have elapsed. This is the best way to have a delay in a mutitasking program because it requires no computer time. By contrast, a FOR/NEXT loop delay uses computer time (best used by other tasks) and does nothing useful. Note that WAIT is a multitasking statement, and as such will not operate unless interrupts are enabled and a source of tics exists (see the chapter on multitasking). When a WAIT statement is executed, interrupts are re-enabled (i.e., an INTON is simulated), since a WAIT without interrupts will never terminate. For example:

```
10 INTEGER I
20 RANDOMIZE
40 I=RND
40 IF I< 0 THEN I=-I
50 WAIT RND/100
60 PRINT "Still rolling along"
70 GO TO 20
```

WATCHDOG

The watchdog timer circuit will perform a hardware reset on the board if all of the following conditions are true:

1) If the watchdog timer jumper JP1 is put in the enable postition "EN".

2) If a MTBASIC program is executing which has been compiled after issuing the direct command NOERR.

3) If the statement WATCHDOG hasn't been executed in more than 0.8 seconds.

Care should be taken not to put a WATCHDOG statement within an interrupt service routine. The reason being, the main program could be crashing and not executing its WATCHDOG command(s) while the interrupt service routine executes its WATCHDOG command upon each interrupt.

Example:

```
10 INTEGER X,Y
20 REM THIS LOOP WILL TAKE MORE THAN 0.8 SECONDS
30 PRINT "LOOPING...";
40 FOR X = 1 TO 10000
50 WATCHDOG
```

```
60 NEXT X
70 PRINT "FINISHED."
```

Type in the program and save it. Now enter NOERR and RUN. The program should execute for a while then finish. List the program then delete the WATCHDOG statement. Now enter RUN and the board should do a hardware reset after about 0.8 seconds.

WCLEAR

WCLEAR erases the currently selected window. The entire window is erased, so if a window is created, framed (with WFRAME) and then cleared, the frame will also be removed, unless the window is resized after being framed. This example creates a window, frames it, and clears it (for more information, see the chapter on windowing).

```
10 WSELECT 1
20 WINDOW 0,0,10,10
30 WFRAME "_", "|"
40 WAIT 500
50 WCLEAR
```

WFRAME <horizontal character>, <vertical character>

WFRAME draws a frame around a window. A frame is simply an outline to give a clear depiction of the window's borders. The two arguments specify the characters to outline the window with. MTBASIC doesn't support true graphics, since graphic controllers are different on all systems, so these two characters simulate a graphics box around the window. The first argument, <horizontal character>, is used to draw the upper and lower window borders. The vertical character is used to draw the left and right window borders. The preferred characters are an underscore ("_") for the horizontal character and the vertical line ("|") for the vertical character. Of course, any other character may be used.

WFRAME draws the frame inside of the window, and so it actually occupies space in the window. It is often a good idea to frame a window but have the frame outside of the borders of the window, so that output directed to the window will not run into the border and so the frame won't be erased by a WCLEAR. This is easy to do. Define a window which is one character larger in all directions than the required window. Frame it, then redefine the window to the needed size. For example, the following program generates a window and then frames outside of its borders. The usable window size is 10 x 10 characters (for more information, see the chapter on windowing).

```
10 WSELECT 1
20 WINDOW 10,10,20,20 : REM Draw a window 1 character too large.
30 WFRAME "_", "|" : REM Frame it.
40 WINDOW 11,11,19,19 : REM This defines the actual window.
50 PRINT "*": REM Put something in the window.
```

WINDOW <UL row>, <UL column>, <LR row>, <LR column>

The WINDOW statement defines the size of a window. A window may not be used until it is both selected (via WSELECT) and defined (via WINDOW). The minimum size for a window is 2 by 2 characters. The first two arguments define the upper left (UL) row and column of the window, while the second two arguments define the lower right (LR) row and column. A window is completely specified by designating its upper left and lower right corners. MTBASIC counts rows from 0 (top of the screen) to 23 (bottom of the screen) and columns from 0 (left hand side of the screen) to 79 (right hand side of the screen). If the user attempts to define a nonsense window (for instance, LR row less than UL row), MTBASIC will set a default window size.

Once a window is defined and selected, all console output will go to that window until another window is selected. MTBASIC will prevent access to any part of the screen outside of the borders of the window (for more information, see the chapter on windowing).

Example:

```
10 WSELECT 0: REM Always select a window first
20 WINDOW 5,20,10,40
30 PRINT "HELLO"
```

WPOKE <expr 1>,<expr 2>

WPOKE acts like a 16 bit POKE. It places the value of <expr 2>, which must be from 0 to 65535 or -32768 to 32767, into the memory location <expr 1>. The following program places 1234 at $8700.

```
10 INTEGER K
20 WPOKE $8700,1234
30 STOP
```

WRSCLK <9 element integer array>

WRSCLK takes the data from the array and writes it to the optional real time clock (part # E010-3). The integer array elements and the data that goes in them are as follows:

element	description of data	range
0	hundredths of seconds	0 - 99
1	seconds	0 - 59
2	minutes	0 - 59
3	hours	1 - 12 (12 hr mode), 0 - 23 (24 hr mode)
4	day	1 - 7
5	date	1 - 31
6	month	1 - 12
7	year	0 - 99
8	mode	0 - 2 (0=pm, 1=am, 2=24 hr mode)

When changing the mode from 12 hour to 24 hour and vice versa, you must correct the hour, but after that the real time clock will keep the correct hour for the selected mode.

Example:

```
10 REM THIS PROGRAM ALLOWS YOU TO SET THE REAL TIME CLOCK
20 INTEGER CLOCK(8),X
30 STRING I$(6)
40 ERASE
50 RDSCLK CLOCK(0)
60 WSELECT 1
70 WINDOW 0,0,80,25
80 CURSOR 0,0
90 X= - 1
100 PRINT "HUNDREDS ";CLOCK(0);
110 GOSUB 1000
120 PRINT "SECONDS ";CLOCK(1);
135 GOSUB 1000
140 PRINT "MINUTES ";CLOCK(2);
150 GOSUB 1000
160 PRINT "HOURS ";CLOCK(3);
170 GOSUB 1000
180 PRINT "DAY ";CLOCK(4);
190 GOSUB 1000
200 PRINT "DATE ";CLOCK(5);
210 GOSUB 1000
220 PRINT "MONTH ";CLOCK(6);
230 GOSUB 1000
240 PRINT "YEAR ";CLOCK(7);
250 GOSUB 1000
260 PRINT "MODE = ";CLOCK(8);
270 GOSUB 1000
280 WRSCLK CLOCK(0) : REM WRITE THE NEW DATA
290 ' OUTPUT LOOP
300 RDSCLK CLOCK(0) : REM NOW READ IT BACK
310 FOR X=0 TO 8
320 CURSOR X,9
330 PRINT CLOCK(X);"      "
```

```
340 NEXT X
350 CURSOR 10,0
360 PRINT TIME$
370 GOTO 350
1000 ' PUTS A NEW VALUE IN THE ARRAY IF A NUMBER IS TYPED AND
1001 ' KEEPS THE ORIGINAL VALUE IF NUL STRING IS ENTERED
1005 X = X + 1
1006 PRINT " ->";
```

program continued on next page...

```
1010 INPUT I$
1020 IF I$ = "" THEN RETURN
1030 CLOCK(X)=VAL(I$)
1040 RETURN
```

WSAVE <integer array>

WSAVE is used to save the contents of the currently selected window to an integer variable array. This feature, coupled with WUPDATE (which restores a window from an array), allows the user to generate pop up menus and save windows to disk.

WSAVE requires an integer array as an argument. Strings may not be used. In MTBASIC, arrays must be specified with an argument, so generally it is best to specify the array with a subscript of 1 (for example T(1)). WSAVE saves all printable characters found in the specified window to the array. The first word (2 bytes) will contain the number of characters saved. The characters will then be packed two to a word into the array. A carriage return will be saved after the rightmost character in each line (remember, spaces are characters too). The array should therefore be dimensioned (via the INTEGER statement) to (the number of expected characters) divided by (2) + 1 + the number of rows. MTBASIC, in the interest of speed, does not check for subscript overflow while filling the array during WSAVE, so it is a good idea to make the array a little on the large size. An array size of 1000 will permit saving the entire screen (for more information, see the chapter on windowing).
Example:

```
10 INTEGER T(1000)
20 WSELECT 1
30 WINDOW 10,10,20,20
40 WFRAME "_", "|"
50 WAIT 500
60 WSAVE T(1)
70 ERASE
80 WUPDATE T(1)
90 WAIT 500
```

WSELECT <window number>

WSELECT is used to enable the windowing system and to specify which (of up to 10) windows is to be used. The argument is an integer from 0 to 9 which specifies the window number. If this argument is more than 9, then windows will be turned off for that task. To disable windowing, WSELECT a <window number> greater than 10.

WSELECT must be issued before any other windowing statements are executed. If a selection is not made, MTBASIC will ignore the windowing statements.

WTIMER <timer#>,<mode>,<count>

This statement sets the registers that are associated with the 64180 microprocessor's programmable reload timers (PRTs). The value for <timer#> can be 0 or 1 which selects TIMER 0 or 1. The value of <count> will be loaded into the timer's reload or data registers, depending on the value of <mode> which is defined as follows:

BIT#	**VALUE**	**ACTION**
0	0	stop timer <timer#>
	1	start timer <timer#>
1	0	don't change data registers

	1	load data registers with <count>
2	0	don't change reload registers
	1	load reload registers with <count>
3-15		not used

The following program toggles the output on pin 20 of HDR4 with each TIMER1 interrupt.

```
10 INTEGER X,DUMMY,MOD,CNT
20 INTON
30 VECTOR 4,1
40 STORMASK 16
50 SETINTS  16
60 PRINT "Enter mode, and count for timer 1"
70 INPUT MOD,CNT
80 WTIMER 1,MOD,CNT
90 GOTO 60
100 TASK 1
110 !THE NEXT 2 LINES RESET THE INTERRUPT FLAG
120 DUMMY = INPORT($10)
130 DUMMY = INPORT($14)
140 PTAOUT (BXOR(PTAIN,2))
150 EXIT
```

WUPDATE <integer array>

WUPDATE is the converse of WSAVE. It redraws the entire window and its contents from the integer array, which must have been saved via a WSAVE statement. As in WSAVE, the argument must be an integer array, and should be specified as, for example, T(1). See WSAVE for an example of the use of WUPDATE and for more information, see the chapter on windowing.

Chapter 6: Functions and User Defined Functions

Functions are called by referencing them in an expression. In MTBASIC, each function returns an INTEGER, REAL, or STRING argument. These are the MTBASIC functions:

ACOS	Arccosine
ADCIN	Returns the digital representation of the selected analog input
ADR	Returns variable address
ASC	Returns ASCII value
ASIN	Arcsine
ATAN	Arctangent
BAND	Bitwise AND
BOR	Bitwise OR
BXOR	Bitwise exclusive OR
CHR$	Returns string equivalent
CONCAT$	Concatenates two strings
COS	Cosine
DATE$	String representation of the real time clock's date
EEPEEK	Returns a 16 bit word from an EEPROM address
ERR	Returns error numbers
ERR$	Returns error messages
EXP	Compute e**x
GET	Returns one character from the current device
INPORT	Reads input expression from input port
INTMASK	Returns the value of the interrupt mask
KEY	Returns one ASCII value from console
LEN	Returns the length of a string
LOC	Returns the value of the internal record pointer
LOG	Natural log (base e)
MID$	Returns part of a string
PEEK	Returns the contents of a memory address
PTAIN	Returns the byte that was written to port A
PTBIN	Inputs a byte from parallel port B
QDIP	Returns the value of the DIP switch emulator
RND	Generates random numbers
RTIMER	Reads the selected 64180 PRT
SIN	Sine
SQR	Square root
STR$	Converts numbers to strings (converse of VAL)
TAN	Tangent
TIME$	The string representation of the real time clock values
VAL	Converts strings to numbers (converse of STR$)

In the following descriptions, the format of the instruction will be shown followed by the type of data that the function returns, which will be STRING, REAL or INTEGER.

ACOS (<expr>)

REAL

Calculates the arccosine of <expr>, and returns this value in degrees.

Example:
```
        10 REAL K
        20 FOR K=1 TO 10 STEP .25
        30 PRINT ACOS(K)
        40 NEXT K
```

ADCIN(<expr>)

INTEGER

This returns the digital value of the analog signal applied to the A/D convertor channel selected by <expr>. The value returned by channels 0 to 7 will be in the range 0 to 1023 for the 10 bit A/D convertor. If the 12 bit A/D option is installed, channels 8 to 15 will return a value in the range of 0 to 4095. Channels 0 to 7 can only be used with the 10 bit A/D and channels 8 to 15 can only be used with the 12 bit A/D option. Channels 0 to 7, or 8 to 15 correspond to the odd numbered pins 1 to 15 of HDR7.

Example:
```
        10 INTEGER CHANNEL
        20 FOR CHANNEL = 0 TO 7
        30 PRINT "CHANNEL ";CHANNEL;" = ";ADCIN(CHANNEL)
        40 NEXT CHANNEL
        50 WAIT 80
        60 ERASE
        70 GOTO 20
```

ADR(<variable name>)

INTEGER

Returns the address of the variable <variable name>. This is typically used to POKE an assembly language routine into an array whose address can be determined with ADR.

Example:
```
        10 INTEGER K
        20 PRINT "Variable 'K' is at ";
        30 FPRINT "H4",ADR(K)
```

ASC(<sexpr>)

INTEGER

Returns the ASCII equivalent of the first character in the string <sexpr>. ASC is the converse of CHR$.
Example:

```
        10 STRING A$
        20 PRINT "Type something..."
        30 INPUT A$
        40 PRINT "ASCII value of the first character is ";
        50 PRINT ASC(A$)
```

ASIN(<expr>)

REAL

Calculates the arcsine of <expr>, which is returned in degrees.

Example:
```
10 REAL K
20 FOR K=1 TO 20 STEP .25
30 PRINT ASIN(K)
40 NEXT K
```

ATAN(<expr>)

REAL

Calculates the arctangent of <expr>, and returns this value in degrees.

Example:
```
10 REAL K
20 FOR K=1 TO 10 STEP .25
30 PRINT ATAN(K)
40 NEXT K
```

BAND(<expr 1>,<expr 2>)

INTEGER

Logically ANDs <expr 1> and <expr 2> as 16-bit integers. A bitwise AND is performed. Each of the 16 bits are individually ANDed.

Example:
```
10 INTEGER I,J
20 FOR I= 1 TO 10
30 FOR J = 1 TO 10
40 PRINT I;" ANDed with ";J;" = ";BAND(I,J)
50 NEXT J
60 NEXT I
```

BOR(<expr 1>,<expr 2>)

INTEGER

Logically ORs <expr 1> and <expr 2> as 16-bit integers. A bitwise OR is done. Each bit is individually ORed.

Example:
```
10 INTEGER I,J
20 FOR I=1 TO 10
30 FOR J=1 TO 10
40 PRINT I;" ORed with ";J;" = ";BOR(I,J)
50 NEXT J
60 NEXT I
```

BXOR(<expr 1>, <expr 2>)

INTEGER

Logically XORs <expr 1> and <expr 2> as 16-bit integers. A bitwise exclusive OR is done.

Example:
```
      10 INTEGER I,J
      20 FOR I=1 TO 10
      30 FOR J=1 TO 10
      40 PRINT I;" XORed with ";J;" = ";BXOR(I,J)
      50 NEXT J
      60 NEXT I
```

CHR$ (<expr>)

STRING

Returns the ASCII character selected by the integer <expr>. <Expr> must be between 1 and 255. CHR$ is the converse of ASC.

Example:
```
      10 INTEGER K
      20 FOR K=33 TO 126: REM printable characters
      30 PRINT CHR$(K);
      40 NEXT K
      50 PRINT
```

CONCAT$ (<sexpr 1>, <sexpr 2>)

STRING

Concatenates two strings into one string; <sexpr 2> is appended after <sexpr 1>.

Example:
```
       10 STRING A$,B$,C$(60)
       20 PRINT "Enter part of a string ";
       30 INPUT A$
       40 PRINT "Now enter the other half ";
       50 INPUT B$
       60 C$=CONCAT$(A$,B$)
       70 PRINT "Here's the whole string"
       80 PRINT C$
```

COS(<expr>)

REAL

Calculates the cosine of <expr>, which must be in degrees.

Example:
```
      10 REAL K
      20 FOR K=1 TO 10 STEP .25
      30 PRINT COS(K)
      40 NEXT K
```

DATE$

STRING

DATE$ returns an eight character string containing the current date from the optional real time clock (part # E010-3). The format is (mm/dd/yy), where mm = month, dd = day, and yy = year.

Example:
```
      10 PRINT "THE DATE IS "; DATE$
```

EEPEEK (<expr>)

INTEGER

Returns the 16 bit data stored at <expr> in the EEPROM. Valid values for <expr> are 0 - 63, so the upper 10 bits will be masked off if <expr> is greater than 63 or less than zero. If <expr> is 0 then the value returned will be 1 if data okay, or 0 if CRC error detected.

Example:
```
      10 INTEGER EEADR
      20 FOR EEADR = 0 TO 63
      30 PRINT " EEPROM ADDRESS ";EEADR;" EEPEEK(EEADR)
      40 NEXT EEADR
```

ERR

INTEGER

ERR is used during disk and printer I/O operations to return the value of the last error generated. If no errors were encountered during the last operation, then ERR returns 0. ERR returns the status of the last I/O operation. If using multitasking, and several tasks are doing I/O, it may not be possible to determine the source of the last error. Interrupts may be disabled (via INTOFF) around the I/O to eliminate this problem. ERR should always be used during I/O to insure that errors are caught. The following errors are returned:

00 - no error
01 - illegal file number
02 - illegal file name
03 - file already open
04 - file not open
05 - no directory space
06 - no disk space
07 - file does not exist
08 - read of write file
09 - write to read file
10 - seek past end of disk, or end of file.
11 - printer not ready
12 - no paper
13 - corrupt data

Example:
```
       10 OPEN 0,1,"FILE"
       20 IF ERR=0 THEN 40
       30 PRINT "File error ",ERR
       40 STOP
```

ERR$

STRING

ERR$ is the counterpart of ERR. ERR$ returns a string containing the error message resulting from the last disk or printer operation. If there was no error, a blank string is returned. The error messages are identical to the table of errors listed above, except for the case where no error is found.

Example:
```
      10 OPEN 0,1,"FILE"
      20 PRINT ERR$
```

EXP(<expr>)

REAL

Computes e**<expr>. A number can be raised to another power by:

```
EXP(LOG(X)*Y): rem (=x**y)
```

Example:

```
       10 REAL K
       20 FOR K=1 TO 10 STEP .25
       30 PRINT EXP(K)
       40 NEXT K
```

GET

INTEGER

Returns one character from the current device. INPUT and INPUT$ read an ASCII line terminated by a carriage return, so they can't read binary files. Because GET only reads single characters, it can read binary files.

Example:

```
      5 ! READ A RECORD FROM A BINARY FILE
      10 INTEGER I,T(150)
      20 OPEN 0,1,"BINARY.FILE"
      30 DEVICE 1
      40 FOR I= 1 TO 128
      50 T(I)=GET
      60 NEXT I
      70 CLOSE 1
```

INPORT (<expr>)

INTEGER

Reads input port <expr> returning an 8 bit value.

Example:

```
      10 INTEGER P
      20 PRINT "Which port (0-65535)?"
      30 INPUT P
      40 PRINT "I/O port # ";P;" holds a ";INPORT(P)
      50 GOTO 20
```

INTMASK

INTEGER

Returns the mask status (which was set by STORMASK) of the 11 maskable interrupt sources.   An 11 bit value is returned and a set bit means the corresponding interrupt will be enabled once an EXIT statement is executed. For more information, see the chapter on interrupts. Below is a table showing which bit of the returned value corresponds to which interrupt.

bit #	**name**	**bit#**	**name**
0	int0	7	COM 1 Tx
1	int1	8	COM 1 Rx
2	int2	9	COM 0 Tx
3	timer0	10	COM 0 Rx
4	timer1	(11-14 not used)	
5	dma0	15	CSI/O

The program below tells which interrupts will be enabled after an interrupt service routine is EXITed.

Example:

```
10 INTEGER ENABLED,INTNUM,ANDMSK
20 ANDMSK = 1
30 FOR INTNUM = 0 TO 10
50 ENABLED = BAND(INTMASK,ANDMSK)
60 PRINT " INTERRUPT #";INTNUM;" WILL BE ";
70 IF ENABLED THEN PRINT "ENABLED" : GOTO 100
80 PRINT "DISABLED"
90 REM MOVE ANDMSK TO THE NEXT BIT POSITION
100 ANDMSK = ANDMSK * 2
110 NEXT INTNUM
```

KEY

INTEGER

Returns the ASCII value of the character the keyboard currently has ready. KEY returns a 0 if no character is ready. Note that the console is the keyboard read, regardless of which device number is active. KEY returns an integer representation of the character (i.e., its ASCII value), so if a string is needed, convert it with CHR$. KEY reads the current character from the keyboard and resets the console's UART. Therefore, the following construct will not work, since when the second KEY is executed, the character is already gone (having been read by the first KEY).

WRONG
```
        10 IF KEY=0 THEN 10
        20 PRINT CHR$(KEY)
        30 GO TO 10
```

Instead, set a variable to KEY and test and print the variable, as in the example.

```
        10 INTEGER L
        20 L=KEY
        30 IF L=0 THEN 20
        40 PRINT CHR$(L)
        50 GO TO 20
```

LEN(<sexpr>)

INTEGER

Returns the length of the string argument <sexpr>.

Example:
```
        10 STRING A$
        20 PRINT "Enter a string"
        30 INPUT A$
        40 PRINT A$;" has ";LEN(A$) ;" characters."
```

LOC

INTEGER

The LOC function returns the value of the internal record pointer. If the record has been defined with a FIELD statement and then a file is opened, LOC returns the next available record number. For example, LOC will return 0 after opening an empty file. LOC will return 3 after opening a file with records 0,1 and 2 already written. This is a convenient way of determining the number of records in the file when it is opened.

```
10 INTEGER X
20 STRING A$
30 FIELD 2,X,9,A$: ' define rec before OPENing if you want to use LOC after.
40 OPEN 2,1,"test.dat": ' open device 1 for random i/o
50 IF  ERR THEN  PRINT ERR$: STOP '  check for error on open
60 PRINT "THE NUMBER OF RECORDS IS "; LOC
```

LOG(<expr>)

REAL

Calculates the natural logarithm (base e) of <expr>.

Example:
```
       10 INTEGER K
       20 FOR K=1 TO 20
       30 PRINT LOG(K)
       40 NEXT K
```

NOTE: To calculate LOG base 10 of a REAL number Z, use: `LOG(Z)/LOG(10.0)`

MID$(<sexpr>, <expr 1>, <expr 2>)

STRING

Returns <expr 2> characters of <sexpr> starting at character <expr 1>.

Example:
```
       20 STRING A$,B$
       30 A$="This is a string"
       40 B$=MID$(A$,3,2)
       50 PRINT B$
```

PEEK(<expr>)

INTEGER

Returns the 8 bit contents of memory address <expr>.

Example:
```
      10 POKE $80,1
      20 PRINT PEEK($80)
```

QDIP

INTEGER

In other revisions of MTBASIC this function used to return the value of the DIP switch. To maintain compatibility with other EPACs, since there isn't a DIP switch on the board, it returns the 16 bit value of a variable from EEPROM. This allows the function to still be useful in cases where QDIP was used to define something such as an identification number. For example, to set the EEPROM variable to 12, run the short program: `10 EEPOKE ($B05),12` . After the board is reset or during power-on reset, the value that was stored will be reflected by the QDIP function. Do not change this value often, because you can only write to the EEPROM 10,000 times.

Example:

```
      10 PRINT "THE BOARD ID # IS "; QDIP
```

RND

INTEGER

Generates a pseudo-random number between -32,767 and 32,767. It's a good idea to reseed the random number generator at the start of your program by executing the RANDOMIZE statement.

Example:
```
       10 INTEGER K
       15 RANDOMIZE
       20 PRINT "Here are 20 random numbers "
       30 FOR K=1 TO 20
       40 PRINT RND
       50 NEXT K
```

RTIMER (<timer>)

INTEGER

RTIMER returns the value of the timer data registers of the timer number indicated by <timer> which can be 0 or 1. It returns a value from -32768 to 32767. The following example will show the value of the source of tics (TIMER0)

Example:
```
      10 PRINT RTIMER (0)
      20 GOTO 10
```

SIN(<expr>)

REAL

Calculates the sine of <expr>, which must be in degrees.

Example:
```
       10 REAL L
       20 FOR L=1 TO 20 STEP .25
       30 PRINT SIN(L)
       40 NEXT L
```

SQR(<expr>)

REAL

Calculates the square root of <expr>.

Example:
```
      10 REAL M
      20 PRINT "Enter a number ";
      30 INPUT M
      40 PRINT "The square root of ";M; " is ";SQR(M)
```

STR$(<expr>)

STRING

Converts the number given as its argument <expr> to a string. STR$ is the converse of VAL.

Example:
```
      10 REAL K
      20 K=1.23
      30 PRINT STR$(K)
```

TAN(<expr>)

REAL

Calculates the tangent of <expr>, which must be in degrees.

Example:
```
       10 INTEGER K
       20 FOR K=1 TO 10 STEP .25
       30 PRINT TAN(K)
       40 NEXT K
```

TIME$

STRING

TIME$ returns a 15 character string containing the current time from the optional real time clock (part # E010-3). The format of the string is as follows:

HH:MM:SS.ss xM_ (in 12 hour mode)
HH:MM:SS.ss 24H (in 24 hour mode)

Where HH = hours, MM = minutes, SS = seconds, ss = hundredths of seconds, x = 'A' or 'P' and _ = space character.

Example:
```
      10 INTEGER NOTYET
      20 STRING WAKEUP$,HRSMIN$
      30 NOTYET = 1
      40 PRINT " IT IS ";TIME$
      50 PRINT " ENTER THE TIME YOU WANT TO WAKE UP"
      60 PRINT " FORMAT HH:MM"
      70 INPUT WAKEUP$
      80 START 1
      90 PRINT "...  SLEEPING ..."
      100 IF NOTYET THEN 90
      110 PRINT "WAKE UP, IT'S "; TIME$; " ALREADY"
      120 GOTO 110
      200 TASK 1
      205 REM EXTRACT THE HH:MM PORTION OF TIME$
      210 HRSMIN$ = MID$(TIME$,1,5)
      220 IF HRSMIN$ = WAKEUP$ THEN NOTYET = 0
      230 EXIT
```

VAL(<sexpr>)

REAL

Returns the numeric value of the number at the beginning of the string expression given as VAL's argument. VAL is the converse of STR$.
Example:
```
       10 STRING A$
       20 PRINT "Type a number"
       30 INPUT A$
       40 PRINT "The value is ";VAL(A$)
```

WPEEK (<expr>)

INTEGER

Returns the 16 bit contents of memory address <expr>.

Example:

```
10 INTEGER X,ADDRESS
20 ADDRESS = ADR(X): REM ADDRESS IS WHERE X VARIABLE IS
30 FOR X = 0 TO 32760 STEP 20
40 PRINT WPEEK(ADDRESS) : REM THIS WILL PRINT THE VALUE AT ADDRESS
50 NEXT X
```

User Defined Functions

MTBASIC supports multi-line user defined functions. A function definition may be any number of lines long. All functions must be defined as follows:

```
10 DEF <function name> {<arguments>}
20 <function definition>
30 FNEND
```

DEF indicates the start of a definition. FNEND indicates the end of a definition. The function name can be up to seven characters, following the rules for variable names. The function name must be declared in an INTEGER, REAL, or STRING statement. The function can have arguments, which are part of the DEF statement, but are not part of the INTEGER, REAL or STRING statement where the function is declared. The arguments are true variables and must be declared. When the function is called, the parameters passed to the function will be copied to these variables so they should have unique names.

User Defined Function rules

Functions must be defined BEFORE they are used. It's a good idea to put all function definitions near the beginning of your program, after the variable definitions. If a function is referenced before it is declared, a FUNCTION ERROR will result. A maximum of 64 functions can be declared in one program. Function definitions cannot be nested. A FUNCTION ERROR will result if a DEF statement is found inside of a function definition.

Function names must be unique. Do not use other variable names or names of MTBASIC statements or functions (even a function name very close to an MTBASIC reserved word may not be acceptable). Arrays cannot be used as arguments to functions. Only simple strings, reals or integers are legal.

Do not attempt to perform console inputs or outputs inside of a function if it will be used in a multitasking program. Your program may hang up since MTBASIC blocks console access during some multitasking operations.

Functions are not recursive. A function cannot call itself, either directly or indirectly.

More on User Defined Functions

The result of a function can be assigned to the function name. To do this, in the function definition use an assignment statement to place the desired value in the function's name (ie. reference the function name like it was a variable name). In the assignment statement, do not specify the function's arguments on the left hand side of the "=" sign. For example, the following function returns the value 1:

```
10 INTEGER FN1
20 DEF FN1
30 FN1=1
40 FNEND
50 PRINT FN1:! The value 1 will be printed.
```

The following function returns the sum of its arguments:

```
10 INTEGER FN1,A,B,C
20 DEF FN1(A,B,C)
30 FN1=A+B+C
40 FNEND
50 PRINT FN1(1+2+3):! The value 6 will be printed.
```

Note that in lines 10 and 30 the function is referred to without its arguments.

Here's another example. This function returns the left N characters of a string:

```
10 STRING LEFT$(127), A$(127)
20 INTEGER N
30 DEF LEFT$(A$,N)
40 LEFT$=MID$(A$,1,N)
50 FNEND
60 PRINT LEFT$("ABCDEF",3):! This prints "ABC"
```

One function can reference another. For example:

```
10 INTEGER FNA, FNB
20 DEF FNB : FNB=1 : FNEND
30 DEF FNA : FNA=FNB : FNEND
40 PRINT FNA:! This prints "1"
```

If functions are using strings, and functions call each other, the "STRING SPACE EXCEEDED" error can result if too many functions have partial string results stored in internal temporary storage. If the message appears, you've called too many functions that need intermediate string storage. Simplify your code somewhat.

Chapter 7: Making Your Program Run Faster

BASICS

MTBASIC attempts to be most things to most people. As a result, some of the tradeoffs made when a program is compiled result in the program not running as fast as it possibly could. This chapter shows how to work around these constraints and maximize the efficiency of a program.

NOERR

Probably the biggest speed improvement is achieved by compiling using the NOERR command (described in the chapter on direct commands). NOERR instructs the compiler not to insert error checking software where it would otherwise be (for example, in checking for SUBSCRIPT-OUT-OF- RANGE). Specifying NOERR before a program is compiled will result in an increase in efficiency of several hundred percent. However, beware of using NOERR on programs which are not fully debugged. It is quite possible that a buggy program will trash compiler memory.

ARRAYS

Whenever possible, use arrays with single dimensions. Two dimensional arrays generate relatively complex threaded code. Singly dimensioned arrays are highly optimized by MTBASIC. Similarly, string arrays are slower than individual strings. Array processing tends to be slower than computing simple variables. The statement A=F(I)+F(I) is inefficient. Instead, use B=F(I): A=B+B. This will result in faster code, particularly if it is in a large loop.

VARIABLES AND CONSTANTS

MTBASIC requires the user to define the mode of all variables used in the program, resulting in much more efficient execution and compilation of the program. Whenever a REAL is assigned to an INTEGER variable or an INTEGER to a REAL, MTBASIC automatically converts the mode of the expression (see Chapter 3 for more on mode conversions). The same process must take place when evaluating constants. Remember, REAL constants are denoted by a decimal point anywhere in the constant (except for binary constants). If the real variable A must have the value 1 associated with it, and the statement A=1 is entered, every time this statement is executed MTBASIC must convert the constant 1 (an INTEGER constant due to the lack of a decimal point) to REAL before assigning it to the variable. This is inefficient. The user should put a decimal point in the constant (1.0) to make it REAL, and avoid the mode conversion. Constants which are used often in the program should be stored in a variable once, and the variable referenced from then on. This eliminates much of the constant evaluation overhead which would otherwise be required, and also reduces the amount of code generated by the compiler.

INTEGER arithmetic is very fast; REAL math tends to be slow, and string processing may be anywhere between INTEGERs and REALs, depending upon the length of the string.

Chapter 8: Windowing

BASICS

MTBASIC provides complete windowing capabilities for the programmer, but this feature can be ignored. Unless specific window commands are issued, MTBASIC functions in a conventional, non-windowing mode.

A window is a subsection of the CRT screen. Any window can be any size up to full screen (24 by 80). Whenever a program selects a window and sends output to it, all output will go to that window. A window's borders are barriers to the PRINT and FPRINT statements, and inhibit these statements from writing anywhere but within the currently selected window.

Note that windows do not scroll. When output runs into the bottom of the window, it will stay there until the cursor is repositioned or the window is cleared (via WCLEAR).

The windowing statements, which are described completely in the chapter on statements, are:

ERASE	erases the CRT.
CURSOR	repositions the cursor within a window.
WCLEAR	erases the selected window.
WFRAME	draws an outline around a window.
WINDOW	defines the size of a window.
WSAVE	saves the contents of a window to a variable, for a later update of the window.
WSELECT	specifies which of the ten windows to use, or returns to non-windowing mode.
WUPDATE	restores a window saved via WSAVE.

MTBASIC defaults to the ADM-3A format, but if you should need to change to a different terminal format contact EMAC for instructions.

Once MTBASIC has been configured, configure your terminal. Many terminals will automatically perform a carriage return at the end of a line, and some will line feed at the end of the page. These features must be disabled. Look in your terminal manual. Typically, these features may be named "auto-wrap" or "auto-newline". If your terminal doesn't allow you to disable these features, make sure no window touches the right hand side of the screen (column 79, counting from 0) or the bottom of the screen (row 23). If you have access to a IBM PC or compatible, EMAC has a communications package, ECOM, which emulates the ADM-3A terminal which runs on the PC, providing ADM-3A emulation and key translation, fully supporting the EDIT direct command.

MTBASIC has only as much control of the CRT as you give it. For windowing to work properly, don't manually reposition the cursor; use CURSOR. Some control codes may cause the cursor to move; avoid these. As long as MTBASIC controls the screen through PRINT, FPRINT, or CURSOR statements, windowing will work properly. Some terminals allow special functions, such as highlighting, to be selected by sending an escape sequence. MTBASIC will pass these commands through as long as the sequence consists of the "lead-in character" (ASCII escape, by default) followed by exactly one character.

DEMONSTRATION PROGRAM

The following program generates random windows on the CRT.

```
1  INTEGER WIN,X,Y,X1,Y1,I,J,K
5  RANDOMIZE
10  ERASE
20  PRINT "Draws random windows"
110  GOSUB 1000
115  ERASE
120  GOTO 110
1000  WSELECT 1
1010   X=BAND(RND,$F)
1020   Y=BAND(RND,$3F)
1030   X1=BAND(RND,$F)
1040   Y1=BAND(RND,$3F)
1042   IF  X1 - X < 3 THEN   X1=X + 3
1044   IF  Y1 - Y < 3 THEN   Y1=Y + 3
1050  WINDOW X,Y,X1,Y1
1060  WFRAME "_","|"
1070  WINDOW X + 1,Y + 1,X1 - 1,Y1 - 1
1080  FOR J=X TO X1: FOR K=Y TO Y1
1090  PRINT K;
1100  NEXT K
1110  PRINT
1120  NEXT J
1130  RETURN
```

USING WINDOWS

Never write a program where more than one INPUT, INPUT$, or KEY is active on one terminal at the same time. MTBASIC can't know which window to get the input from if more than one is active, and will give unpredictable results.

When more than one task are using windows at the same time multitasking is altered slightly. Windowing statements cannot be executed in another task if a task that has been interrupted was in the middle of using a windowing statement. This is most noticeable when a large window is being WCLEARed and other tasks are printing data to their windows. To understand what happens internally, let us imagine one task was in the middle of executing a WCLEAR when a tic was received. The next task will execute all of its own non-windowing statements, but if it encounters a windowing statement it will not execute it but will set a window flag (which indicates that another task wanted to use a windowing statement) and do the next available task. If the next available task is using windowing statements, it will respond the same way. Eventually the task that was executing a WCLEAR will start executing again. If another tic occurred before WCLEAR was finished, the same process will start over again, otherwise if the WCLEAR has finished processing, MTBASIC will check the window flag and generate a pseudo tic if the flag was set. Though this slightly alters the timing of multitasking, the flag guarantees that if a task has multiple windowing statements it will not start executing another until the other tasks have been allowed to use their windowing statements.

Framing is a very useful, but easily misunderstood feature. Whenever WFRAME is executed, an outline is drawn within the currently selected window. The outline consists of characters physically inside of the window. If the window size is not changed after a WFRAME, any PRINT directed to the window can overwrite part of the frame. Therefore, after performing a WFRAME, resize the window one character smaller in all four directions. For example:

```
10 WSELECT 5
20 WINDOW 10,10,20,20
30 WFRAME "_", "|"
40 WINDOW 11,11,19,19
50 FOR T=1 TO 500
60 PRINT "*"
70 NEXT T
```

Chapter 9: Recursion

BASICS

MTBASIC is a completely recursive language. By definition, a recursive routine is one that is able to call itself. This may seem a paradox to the uninitiated, but it can be very useful once you understand the logic. The classic example of recursive thinking is the statement "This statement is false". The statement can't be true if it's false or false if it's true, or etc. Recursive programs can be a bit more useful. Also, recursive programs only function if they return after a finite number of calls.

Where would recursion be useful? One common recursive mathematical technique is computation of factorials. Although factorials can be programmed in a loop, it is more elegant to program them recursively. For example, the following MTBASIC program will compute factorials in a recursive fashion.

```
10 REAL A
20 INTEGER I
30 PRINT "Enter number to take factorial of.  It must be"
40 PRINT "bigger than 1 and less than 10."
50 INPUT I
60 A=1.0
70 GOSUB 100
80 PRINT "Answer is ",A
90 STOP
100 A=A*I
110 I=I-1
120 IF I=0 THEN 140
130 GOSUB 100
140 RETURN
```

Other uses of recursion include the processing of linked lists and binary trees. These are all structured in an orderly fashion and determination of the next head or tail is frequently simplest using a recursive routine.

The evaluation of expressions is always done recursively. Consider the case of analyzing a mathematical formula; the routine which processes the formula must be able to call itself so that it can evaluate arguments of functions, subscripts, etc.

RESTRICTIONS

MTBASIC places one restriction on recursive programs: they must not call themselves too deeply. It is difficult to predict exactly how deeply the program may call itself because it depends upon the nature of the program being executed. There has not been problems with programs that call themselves ten times and some programs have called themselves over fifty times. In general, though, we recommend that you limit recursive programs to a depth of ten to prevent the stack from overflowing and causing MTBASIC to crash. If you need a greater depth backup the program on RAMDISK then try it and see what happens. Recursive programs should not be combined with multitasking programs, since both recursion and multitasking use large amounts of the stack.

BASICS

Simple programs written in MTBASIC generally communicate only with the console. As more sophisticated programs are written, it is often necessary to access and store data to a RAMDISK file. This gives the user the ability to manipulate data bases and store data for later operations. Additionally, it is frequently useful to be able to perform I/O to other terminals, as, for example, in an application where it is necessary to enter and view data from one terminal while sending other data to a PC communications package for storage on hard disk. MTBASIC provides the statements necessary to control both files and devices.

When discussing files, it is important to differentiate the two distinct types of file operations: direct file commands and file handling statements. Direct commands are not components of an MTBASIC program. The direct file commands provided in MTBASIC are SAVE, LOAD, DEL, DIR, TYPE, RECEIVE and FORMAT which allow the programmer to store, retrieve and delete programs, list the files, send an ASCII file from RAMDISK to the console, send an ASCII file from the console to the RAMDISK and format a RAMDISK, respectively. These commands are not discussed here (see Chapter 2) and cannot be part of an actual MTBASIC program.

SAVEd files are stored in ASCII and can therefore be altered (or generated) by a text editor, if downloaded to or uploaded from a personal computer using ECOM or some other communications package.

Detailed descriptions of the device handling statements are given in Chapter 5, but their functions will be summarized here. The statements which are related to file and device handling are: OPEN, CLOSE, DEVICE, FIELD, PUT, RGET, PRINT, INPUT, INPUT$, and FPRINT. The GET, ERR and ERR$ functions, described in Chapter 6, are also associated with device handling.

MTBASIC's general approach to performing device I/O is to use the standard PRINT, INPUT, INPUT and FPRINT commands. Normally these commands communicate with COM1 , but their input or output can be redirected to any other file or device using the DEVICE statement.

Before a RAMDISK file is to be handled it must be OPENed. This has the effect of making the file known to the system. An OPEN statement opens the file and associates a "device number" with the file name. In the current version of MTBASIC up to four files may be open at a time. That device number is then used within the program to reference that file. For example, if the OPEN 0,1,"ABC" statement is executed, then DEVICE number 1 is associated with file "ABC". Whenever a file is opened, the user must indicate whether read or write operations will take place to the file. This is specified in the first argument of the OPEN statement (in this case the 0 indicates that "ABC" is to be used for read functions).

Much as the file must be opened to be used, a file must be closed when it is no longer required. Closing disassociates the previously assigned device number from the file name, making that number available for use with other files. CLOSE guarantees that all information will be written to the file.

Once a file has been opened, it is only necessary to execute a DEVICE statement to direct I/O to that file. This tells the system that all subsequent I/O will take place to the device number specified in the argument of the DEVICE statement.

The ERR and ERR$ functions can be used to determine if errors occur during I/O. These functions are only significant during disk file I/O, or printer output, since I/O performed on other devices does not create errors. ERR returns the error number of the error encountered and ERR$ returns a string containing the error message. It is strongly recommended that whenever a I/O operation takes place the ERR function be used to determine what errors, if any, have been detected. The meanings of the error numbers are listed under ERR in Chapter 6.

The program below will prompt the user for a file name and then store the next 30 characters typed at the terminal to that file. After the 30 characters are typed, the file is CLOSEd then OPENed for reading and the characters are read from it and printed to the terminal.

```
10 INTEGER X,Y
20 STRING FILE$
30 PRINT "Enter file name ";
40 INPUT FILE$
50 OPEN 1,1,FILE$:! OPEN FILE$ FOR WRITING
60 IF ERR>0 THEN GOTO 30
```

continued on next page....

```
70 PRINT "Type 30 characters."
80 FOR Y=1 TO 30
90 DEVICE 0:! SELECT DEFAULT DEVICE
100 X=GET:! GET A CHARACTER FROM THE TERMINAL
110 PRINT CHR$(X);:! ECHO IT
120 DEVICE 1:! SELECT RAMDISK FILE
130 PRINT CHR$(X);:! SEND CHARACTER TO FILE
140 IF ERR>0 THEN GOTO 160
150 NEXT Y:! GET NEXT CHAR
160 CLOSE 1:! CLOSE THE FILE
170 DEVICE 0
180 PRINT ERR$:! SHOW ERRORS IF ANY
200 ! NOW PRINT THE DATA FROM THE FILE
210 OPEN 0,1,FILE$:! OPEN FILE$ FOR READING
220 DEVICE 1:! SELECT THE FILE
230 X=GET:! GET A CHARACTER FROM THE FILE
240 DEVICE 0:! SELECT TERMINAL
250 PRINT CHR$(X);:! PRINT TO TERMINAL
260 IF ERR=0 THEN GOTO 220
270 CLOSE 1
```

FILE RECORDS

The FIELD statement can be used to specify the format of a record of data in a RAMDISK file. When a file's record structure is defined with FIELD, then PUT and RGET must be used to transfer data to the file. FIELD defines the record's format, PUT transfers data to the file, and RGET transfers data from the file. The FIELD statement is used to define the exact number of bytes to be transferred for each variable written to or read from a file. FIELD's arguments must be supplied in pairs. The first member of each pair is the number of bytes to be transferred, while the second is the name of the variable to be transferred.

For example,

```
10 FIELD 2,A$,3,B$,4,C$
```

indicates that 9 bytes (2+3+4) are to be transferred. Two bytes of A$ are specified, followed by 3 bytes of B$, and ending with 4 bytes of C$. Unlike most other BASICs, in MTBASIC the arguments to FIELD may be integer, real or string. The arguments are simply variables, and can be used as any variable may. Arrays may not be used in a FIELD statement. If integer variables are specified, 2 bytes should generally be used for each variable. It is, however, possible to use a single byte. Real values require 4 bytes.

EXAMPLE:

```
      10 STRING A$
      20 INTEGER X
      30 REAL Y
      40 FIELD 10,A$,4,Y,2,X
      ...
```

While FIELD specifies the record format, PUT and RGET actually transfer the data. The following program writes 2 bytes of A$ and 3 bytes of B$ to a file.

```
10 STRING A$,B$
20 A$="12"
30 B$="345"
40 OPEN 1,1,"FILE"
50 DEVICE 1
60 FIELD 2,A$,3,B$
70 PUT
80 CLOSE 1
```

Note that PUT and RGET do not take arguments. Each PUT or RGET accesses the next record in the file. If an EOF error occurs following an RGET the data retrieved is undefined.

RANDOM I/O FILES

MTBASIC allows both sequential and random I/O files. By definition a sequential file is one which is accessed in a serial manner. When reading a sequential file, each RGET, GET or INPUT reads sequentially from beginning until the end, (or as far as desired). When writing a sequential file, each PRINT or PUT adds data to the end of the file. By contrast, a random file allows access to any record in any order desired and it also allows interleaving of record reads and writes. Random files are OPENed by setting <flag> to 2 (see OPEN statement).

The function LOC returns the value of the internal record pointer. When a file is just opened, it points to the next available record number. For example, LOC will return 0 after opening an empty file. LOC will return 3 after opening a file with records 0,1 and 2 already written. This is a convenient way of determining the number of records in the file when it is opened.

The statement SEEK(<expr>) allows you to select the record to write or read. SEEK sets the internal record pointer to the record <expr>. If that record has not been written yet, an End of file message will occur (EOF) and the record pointer will point to the file space following the last record in the file (if the file is empty, it will point to the beginning of the file). For example, if you SEEK(32767) and only records 0 to 4 have been written, an EOF will occur and the record pointer will point to the file space following record 4 and after this, LOC will return a value of 5. The combination of SEEK(32767) followed by LOC is an easy way to determine the number of records in a file after the file has been written to.

As described earlier, PUT stores and RGET retrieves records at the current record pointer and both increment the record pointer after doing so. The record pointer is not incremented in the following cases: when during a PUT an Out of Disk Space error occurs, or if the record pointer points to a record that hasn't been written yet (i.e. it points to the end of the file) and an RGET occurs.

Note the following:

Only 1 random file may be open at once.

Only 32768 records are allowed per file (record 0 is the first, 32767 is the last).

It is not necessary to close a random file to retain data as is the case with write files.

If a valid file device number has not been selected before RGET or RPUT, these will be directed to the file allocated to DEVICE 1, if any (the current device is not changed, though). Remember to use a DEVICE statement before RGET or RPUT if the file was OPENed to devices 2, 3 or 4.

This program demonstrates random file I/O. First, records 0 to 4 are written and you are prompted to enter a record number. Any errors that occurred during the SEEKing or during RGET of that record are displayed followed by the data in the record requested. If the record selected hasn't been written yet, the message "record empty" will be seen. Following this the value that LO had after the RGET is displayed. The record selected will be modified and then all the records in the file will be displayed. To demonstrate the operation of SEEK when a record is sought that hasn't been written yet, enter "9" for the first record number.

```
120  INTEGER X,Y,Z
130  STRING B$(22),EE$
140  INTOFF
150  OPEN 2,1,"test.dat": ' open device 1 for random i/o
160   IF  ERR THEN  PRINT ERR$: STOP : ' CHECK FOR ERR ON OPEN
170  FIELD 2,X,11,B$,2,Y: ' define record
180  DEVICE 1
190  SEEK (0)
200  FOR X=0 TO 4
210  DEVICE 0
220  ' make b$ the string representation of x and pad with periods
230   B$=STR$(X):  B$=CONCAT$(B$,"..............")
240   Y=X + 100
250  PRINT X,B$,Y: 'show what we're going to write to the file
260  DEVICE 1: ' select file 1
270  PUT : ' store the record
280  NEXT X
290  DEVICE 0
```

continued on next page....

```
300  PRINT "enter record #"
310  INPUT Z
320   IF  Z < 0  OR  Z > 9 THEN  STOP : ' QUIT IF OUT OF RANGE
330  DEVICE 1
340  SEEK (Z)
350   EE$=ERR$: ' save error message (if any)
360   B$="                      ": ' b$ must be full before loading with rget
370  RGET : ' read record z
380  DEVICE 0
390  PRINT "SEEK ERR$= ";EE$
400  PRINT "RGET ERR$= ";ERR$
410   IF  ERR THEN  PRINT "RECORD EMPTY": GOTO 430
420  PRINT "record ";Z;" = ";X,B$,Y: ' show data in record
430  PRINT "LOC AFTER RGET="; LOC
440   B$="**DATA**": ' CHANGE THE DATA...
450   X=Z:  Y= - Z: ' ...BEFORE WRITING IT BACK
460  DEVICE 1
470  SEEK (Z)
480  PUT : ' change the record
490  SEEK (0): 'point to start of file
500  'seek is not needed in the loop since RGET incs the internal rec pointer
510  FOR Z=0 TO 9
520  DEVICE 1
530   B$="                      ": 'b$ must be full before loading with rget
540  RGET : ' read the record
550  DEVICE 0
560   IF  ERR = 0 THEN  PRINT "REC#";Z,X,B$,Y: ' DISPLAY THIS RECORD IF VALID
570  NEXT Z
580  GOTO 290
```

Chapter 11: Multitasking

BASICS

Multitasking is the process of running more than one activity at apparently the same time. Each activity is called a "task". Of course, it is impossible for more than one instruction to execute on a single processor at any one time, so multitasking simply gives the appearance of running more than one program at a time. This concept is also known as concurrency. The MTBASIC software switches execution between each of the tasks at a high rate of speed; 60 times per second.

A multitasking program consists of two or more tasks which run asynchronously with respect to each other. Unless the programmer uses semaphores to provide some sort of synchronization of execution, it is difficult to tell when any one task will be executing. Tasks are not like subroutines. A subroutine only runs when it is called, and terminates when its return instruction is executed. A task, on the other hand, may be running at any time, as computer time is shared between execution of the tasks.

Chapter 5 describes MTBASIC's multitasking statements in detail. These statements are TASK, START, EXIT, WAIT, CANCEL, VECTOR, INTON and INTOFF. A detailed example of creating a multitasking MTBASIC program is included in this chapter.

INTERRUPTS AND MULTITASKING

Central to the concept of multitasking is the interrupt. An interrupt is traditionally a hardware signal which stops the current execution of the processor and causes the computer to start running something else. When an interrupt is received it is possible to service that interrupt without the executing program ever knowing that an interrupt was received. For example, an interrupt service routine can be written which simply counts interrupts, so that an executing program can read the total number of interrupts that have been received. This is a simple way of implementing a clock.

Interrupts are important to multitasking since they are the mechanism by which one task is stopped and another is started. In a traditional multitasking system, a real time clock (a source of interrupts coming at a predetermined rate - typically 60 per second) interrupts the executing program and causes the system to switch execution to a different task. Execution of each task is sequenced by these interrupts; task 1 may run, then task 2, then task 3, then back to task 1, etc. A particular task generally does not run to completion before the computer starts running another task. The computer just suspends execution of one task and goes to another. Eventually, it picks up with the suspended task where it left off and resumes execution. Although each task is executed in a "choppy" fashion, to the user it appears as if all tasks are executing smoothly together because of the computer's speed.

The interrupt, then, is the basic unit of task switching. Within MTBASIC, interrupts from the 64180's TIMER0 are referred to as tics. Whenever MTBASIC receives a tic, it switches execution of the current task to the next task which is ready to be executed. In a program which contains only a single task (a non-multitasking program) the receipt of a tic does not cause anything to happen since there are no other tasks. In a dual task program, the receipt of each tic causes execution to switch between the two tasks. In a three task program, receipt of each tic causes execution to sequence between the three tasks. The advantage of using hardware interrupts is that they can be designed to generate a very precise frequency. This makes the control of task switching very accurate; the user can specify events to occur (a task to run) at a specific and exact time.

So far we have discussed interrupts only in the context of the tics required for multitasking. MTBASIC also supports one other source of interrupts, device interrupts, which are present in some computer systems. For example, many UARTs generate an interrupt when data is ready to be read from them. MTBASIC allows the user to process this interrupt and start a task whenever a device interrupt occurs. Using this concept, a UART service routine could be written entirely in MTBASIC by dedicating a task to the purpose of servicing the device interrupt. Device interrupt handlers are extremely useful, since the program is not forced to continuously poll a particular device to determine if data is ready. When the device has data ready, it simply sets an interrupt which causes execution of the task to service the device and read the data. The VECTOR instruction, described in Chapter 5, is used to interface an interrupt device handler to MTBASIC.

SCHEDULING

MTBASIC offers powerful statements which allow very sophisticated control of the execution of the tasks. This task execution control is referred to as scheduling, since the user's program may schedule how often any individual task runs.
All MTBASIC programs consist of a main program which is also known as the "lead task". The lead task must start at least one other task for the program to multitask. After another task has been executed, that task may start another or may start several. MTBASIC's START statement is used to start

additional tasks executing.

The simplest form of multitasking consists of a lead task which starts one or more other tasks. Each task then executes constantly. For example, the following program causes each task to print a number. The three tasks run continuously, so the numbers 0, 1, and 2 will be constantly printed out at the console during the execution of these tasks. In this program, the second argument of the START statement, the schedule interval, is not significant, since the tasks never EXIT.

```
20 START 1,100
30 START 2,100
40 PRINT 0
50 GOTO 40
60 TASK 1
70 PRINT 1
80 GOTO 70
90 TASK 2
100 PRINT 2
110 GOTO 100
```

A more sophisticated version of the previous program is shown below. Note that in this program there are no GOTO statements, other than the one in the lead task. Each of the two sub-tasks execute the print statement once, and then perform an EXIT. EXIT ceases the execution of that task. However, since each of these tasks were started with a schedule interval in the START statement, the tasks will be "reborn" after the number of tics specified in the START statement's schedule interval elapses. TASK 1 will print every time 100 tics go by. TASK 2 will run twice as fast as TASK 1 (every 50 tics).

```
20 START 1,100
30 START 2,50
40 GOTO 40
50 TASK 1
60 PRINT "Task 1"
70 EXIT
80 TASK 2
90 PRINT "Task 2"
100 EXIT
```

Variables defined within the program are global to all tasks in the system. This means that every task has access to all of the variables, and therefore each task must be careful not to modify variables used by other tasks. Variables also provide a facility for communication between tasks. Values can be set in a variable in one task and tested in another. For example, the following program, which consists of two tasks, prints out a message whenever task 1 counts 100 tics.

```
20 INTEGER I
30 I=0
40 START 1,1
50 IF I<>100 THEN 50
60 PRINT "100 tics counted"
70 I=0
80 GOTO 50
90 TASK 1
100 I=I+1
110 EXIT
```

USES FOR MULTITASKING

What can multitasking be used for? There are as many uses for multitasking as there are for computers. Here are a few examples. In a process control program it is often useful to assign one task to each process being handled. For example, in a steel mill a system could be installed to measure the thickness of the steel being produced and adjust the mill to produce a particular thickness. Steel making is an ongoing process, so the program should not stop when the operator is entering data. One task could be assigned to reading parameters entered by the operator while another task reads the thickness of the steel. Yet a third task could be responsible for controlling the mill's jack screws to produce the desired thickness. Other tasks may be needed to perform calibrations, to display thickness values on different consoles located throughout the mill, and even to provide financial and historical data on the steel being produced. Closer to home, a control environment exists in a fully instrumented house. For example, it is possible to wire a house to a computer in such a fashion as to discourage even the most dedicated of burglars. An intrusion detection system could be connected to the computer. One task could be responsible for monitoring this intrusion detector and calling the police using an auto-dial modem if a burglar is sensed. Another task could be scheduled to run every hour to turn on the bathroom light for five minutes and then turn it off again, while a third task could be scheduled to run every fifteen minutes and turn the kitchen light on and off. If two terminals exist and MTBASIC's device I/O commands are used, it is possible for two users to communicate to the board at the same time, or one terminal could display information and the other could be used to enter data. Some experimenting with multitasking will give you a better understanding of how to incorporate this MTBASIC feature into your programs.

PRACTICAL CONSIDERATIONS

All MTBASIC programs start with a lead task, which may never cease execution, although it may execute a WAIT statement. The lead task may not execute a STOP or an EXIT. In a nonmultitasking program (a conventional BASIC program), only the lead task exists. All additional tasks (those other than the lead task) must start with a TASK statement, which is used to identify the start of the task to the system. Each of these tasks must end in a GOTO someplace within itself, or an EXIT. One task may not run into another task, since this will stop the entire program. If any task in a system executes a STOP the whole program will stop executing. To start execution of a task (other than the lead task), a START statement must be executed for that task.

What is a good rate for the hardware interrupts? If the interrupts come too slowly, it will appear that one task is running at a time and that the computer is switching between tasks. This is generally an undesirable situation. On the other hand, if the interrupts come too quickly, MTBASIC will spend all of its time switching between tasks and little time actually executing the tasks. This is equally undesirable. Typically, MTBASIC programs run very well with an interrupt rate of 60 hertz or less (the default rate). This is the standard interrupt used on most minicomputer systems, since it provides convenient timing for clock generation. One warning - hardware interrupts which come too fast for MTBASIC to process will cause erratic operation.

NOTES REGARDING INTERRUPTS

The microprocessor allows control of interrupts with the DI and EI instructions. These instructions are implemented in MTBASIC through the INTOFF and INTON statements respectively. The INTOFF statement disables interrupts globally, meaning that even if some interrupt sources are enabled through the SETINTS statement they will not produce interrupts until an INTON statement is executed.

The INTOFF and INTON statements provide convenient methods of turning multitasking on and off within the lead task. At any point within the lead task INTOFF may be executed to stop multitasking. The lead task will remain the only task which executes until an INTON is executed. This gives the user the ability to give all of the processor time to the lead task When the lead task is finished, it must execute an INTON so that the other tasks can continue. The SETINTS and INTMASK statements can be used to give all processor time to a particular task which is not necessarily the lead task.

VECTOR creates links to MTBASIC tasks allowing them to be "interrupt handlers". Once the links are created, they cannot be removed. For example, if a program is run which uses a VECTOR statement to link INT0 to task 2 and then you stop the program and remove the VECTOR statement, the link between INT0 and the internal code that had jumped to task 2 will still be there. MTBASIC will generate a task error and stop the program if an INT0 is received. To prevent this, make sure that all interrupts that are enabled have been linked to a task by the VECTOR statement.

When MTBASIC starts it automatically performs an INTON statement, therefore, DO NOT turn on your hardware interrupts (with SETINTS or STORMASK) until the links have been established by executing a VECTOR.

When an EXIT statement is used within an interrupt handler, it will enable individual interrupt sources according to the value used in the last STORMASK statement. WAIT will perform context switching if another task is ready.

Tasks that are used as interrupt handlers and are not able to be interrupted must run to completion rapidly, so if extensive processing is needed, the interrupt handler should START an additional task to do the processing. Also, do not use PRINT statements within an interrupt handler task or the program will not work properly. If you must use a PRINT statement, then set a flag within the task and have the main task poll the flag and PRINT the desired message or START a task that will PRINT the desired data.

As described in Chapter 5, the START statement starts execution of the indicated task one tic after the START statement, regardless of what the schedule interval is. If a task must start executing at a later time, put a variable in the specified task to act as a flag. The first time that a program is executed, the flag is set to a 1, which allows complete processing of the task on the next invocation of the task. For example, the following program prints a message 200 tics after the START statement for task 1 is executed.

```
10 INTEGER I
20 I=0
30 START 1,199
40 GOTO 40
50 TASK 1
```

```
        60 IF I=0 THEN 80
        70 PRINT "TASK 1"
        80 I=1
        90 EXIT
```

This technique can also be used to increase the schedule interval. The START statement allows a maximum of 32,767 tics to be specified as the schedule interval. This number can be increased to any value by setting a flag which counts the number of times the task has been executed, and allows the task to continue executing only if a certain number of counts have been detected. Before each TASK statement is compiled, MTBASIC puts a STOP in the compiled program. Therefore, if any task attempts to run into another task, the program will stop. This is also true of the last statement in the program; a STOP is placed after the last statement, so that if the program tries to run off the end of the program, it will stop.

The CANCEL statement is used to stop scheduling of a particular task. Normally, a task stops by executing an EXIT statement. The schedule interval specified in the START statement will cause the task to start executing again later. CANCEL stops the scheduling process. If a CANCEL statement is executed for a particular task, the task will not be restarted again, unless another START statement for that task is executed. CANCEL does not kill the task immediately. The task will continue executing until an EXIT statement is encountered or until a context switch occurs. CANCEL only stops the scheduling of the task, not the task itself. CANCEL also provides a facility for running a task only once. The task merely has to CANCEL itself anywhere within the body of the task. Any task may CANCEL any other task, or any task may CANCEL itself. The lead task doesn't have a TASK statement, so it may not be CANCELed, unless a STOP statement is used (STOP cancels all the tasks).

The KEY function is used to read a single character from the keyboard. The KEY device is always the console. If one task executes a KEY function, while another task is in the midst of an INPUT statement from the console, the values returned from KEY and INPUT will be unpredictable. One task will be attempting to "steal" characters from the other.

EXAMPLE MULTITASKING PROGRAM

There may have been times when you needed to write a program that did several things at once. Doing this is not an easy job in most computer languages. MTBASIC programs have the ability to be interrupted so that they can multitask (or do several things at once). Each time the program is interrupted, it switches to some other sub-program (a task) to execute as an overlay of sorts for the main program (lead task). Let's start with a short example of how tasks can be used.

```
10 INTEGER I: I=1: REM Initialize variables
30 START 1,100 : REM START TASK 1 every 100 tics
40 PRINT "MAIN PROGRAM": REM Print a message to the screen
50 GOTO 40: REM Do this infinitely
60 TASK 1 : REM The first TASK starts here
70 PRINT  "INTERRUPT #";I : REM Alert the user of an interrupt
80 I=I+1 : REM Increment counter
90 EXIT : REM Leave this TASK
```

Type this program in and RUN it. You may notice that once in a while (every 100 tics) the usual infinite loop stops and prints "INTERRUPT #" followed by a number. You have just seen multitasking in action. What stopped the computer? It stopped itself. Take another look at the program. Lines 30, 60 and 90 all contain statements unique to MTBASIC. They are a few of the statements that support multitasking and interrupt handling. The START statement tells the program to start executing the task indicated by the first number "1" right now and then to start it again when 100 tics have occurred. Variables can be used for the second number in order to permit changing the scheduling interval for the task. At line 30 the program is set to START a task, but which task? The "1" following START is the task number (in this case there is only one possible task, but there could be as many as 32). Line 60 is the start of task 1. When this task's turn to be executed comes up, this is where the program goes. So far we've seen how to start this task, but how will we get out of it? In this example line 90 (EXIT) is the end of the task.

EXIT works like a RETURN after a GOSUB. The program will continue execution at line 40, but will restart task 1 (at line 90) in another 100 tics. There are also other statements, mentioned earlier in the chapter, which can control the beginning and ending of tasks, so that tasks can run at varying rates or for a conditional number of times.

How are tasks different from subroutines? A task is started only once (via the START statement), but continues to execute, sharing computer time with the main program and other tasks. A subroutine only executes when called, and stops all other activities while it is running. The uses of tasks and subroutines are not really the same.

Chapter 12: Programmable Reload Timers

The MICROPAC 180 comes equipped with a two 16 bit Programmable Reload Timers. Each PRT channel contains a 16 bit down counter and a 16 bit reload register. The down counter can be directly read and written and if the timer counts down to 0 an interrupt can be generated. The timer may be programmed to automatically reload after counting down to 0 and continue counting or it may be programmed to stop counting once it reaches 0.

TIMER CONTROL

MTBASIC provides control over the timers through the WTIMER statement. The format of the WTIMER statement is as follows:

```
WTIMER <timer#>,<control>,<count>
```

The value of <timer#> selects the timer (0 or 1) and <count> is the value that will be loaded into the PRT data and/or reload registers, depending on the value of control which is defined below.

BIT#	VALUE	ACTION
0	0	stop timer <timer#>
	1	start timer <timer#>
1	0	don't change data registers
	1	load data registers with <count>
2	0	don't change reload registers
	1	load reload registers with <count>
3-15		not used

You can read the value of a PRT data register by using the RTIMER(0) or RTIMER(1) functions.

The timers are driven at the system clock speed divided by 20. Therefore if the system clock is 6.144 MHz then timer clock is being driven at 307.2 KHz. The maximum time length that the timers can achieve per interrupt is 213.33 Milliseconds.

Writing directly to the timers causes them to be stopped momentarily, so this will affect the timing slightly. When either of the timers count down to 0 (reach their Termination Count) they are automatically reloaded by the value in their respective reload registers. The reload registers default to a value of FFFFH at reset. For timer I/O address information see the MICROPAC 180 Hardware Reference Manual.

The following examples assume a clock frequency of 6.144

Example 1 : `WTIMER 0,%11,30720`

will cause 30720 to be loaded into TIMER0's Data Registers while not changing the existing values in the Reload Registers (FFFFH after reset). The execution of this command will then cause TIMER0 to start, initially interrupting the processor once after a period of 100 milliseconds, after which TIMER0 will reload the Data Registers from the current values of the Reload Registers and continue counting with the new values.

Example 2 : `WTIMER 1,%111,61440`

will cause 61440 to be loaded into TIMER1's Data Registers and Reload Registers. The execution of this command will then cause TIMER1 to start, initially interrupting the processor after a period of 200 milliseconds, after which TIMER1 will reload the Data Registers from the current values of the Reload Registers (61440) and continue counting with these values. Subsequent interrupts will continue every 200 milliseconds.

Example 3 : `WTIMER 1,%110,3072`

will first cause TIMER1 to stop counting, then 3072 will be loaded into TIMER1's Data Registers and Reload Registers. TIMER1 can then be started at any time by executing the command " WTIMER 1,0,0 " which will start the timer without changing the Data or Reload registers. The processor would then be interrupted every 10 milliseconds, if interrupts have been enabled for TIMER1.

NOTE: For other examples using the timers with interrupts, see the section on interrupts in this manual.

In order for either timer to interrupt the processor their associated interrupts must be enabled. This can be accomplished through the use of the SETINTS statement. See section on interrupts in this manual for further information.

The timers can be used in a polled mode as well as in an interrupt mode. To use the timers in a polled mode simply execute an IN($10),mask bit 6 for TIMER0 or bit 7 for TIMER1 of the Timer Control Register (TCR), I/O address 10H. Bits 6 and 7 of the TCR are set to 1 when their respective data registers decrement to 0. Reading the TCR *and* reading the high or low byte of the associated timer data register will clear the respective bit. For more information see the section on TIMERS in the MICROPAC 180 Hardware Reference Manual. The following is an example program using TIMER1 in the polled mode of operation:

```
10  INTEGER X,Y,DUMMY,COUNT
20  Y=0
30  PRINT "ENTER COUNT FOR TIMER"
40  INPUT COUNT
50  WTIMER 1,%111,COUNT
60  X=INPORT($10)
70  X=BAND(X,128): 'MASK OFF ALL BUT BIT 7
80  IF X=0 THEN GOTO 60
90  Y=Y+1: ' Y IS THE # OF TIMES THE TIMER HAS COUNTED DOWN
100 'CLEAR TIMER FLAG
110 DUMMY = INPORT($10)
120 DUMMY = INPORT($14)
130 IF Y<50 THEN GOTO 60
140 PRINT "THE TIMER HAS COUNTED DOWN 50 TIMES"
150 GOTO 20
```

Chapter 13: Interrupts

MTBASIC supports all 64180 HARDWARE INTERRUPTS both internal and external. Access to INT0 and INT1 external INTERRUPTS is through the EXPANSION CONNECTOR. The INT2 INTERRUPT is tied directly to the A/D converter's Data AVailable (IC ML2230 DAV pin) output; the NMI INTERRUPT input is accessible through screw connector ST1 pin #1, the same connector as used for the RESET input (SEE COMPONENT LAYOUT DRAWING AT THE END OF THIS MANUAL).

All interrupts except the NMI interrupt may be vectored to tasks dedicated to handling these interrupts. The NMI INTERRUPT is predefined (not vectorable) to return to the MTBASIC prompt (">"). This is helpful when trying to break from a program that has been compiled after using the NOERR direct command. The RESET input, available through the same screw connector, does a hardware reset which is the same as turning off then turning on the board (except that volatile memory is not erased).

INTERRUPT TRIGGERS

The INT0, INT1, and INT2 external interrupts are LEVEL SENSITIVE, meaning they are able to be acknowledged by the processor when their signal is held high. The NMI interrupt is RISING-EDGE SENSITIVE, meaning that when a transition from a low-signal to high-signal occurs on the NMI input line the 64180 will acknowledge the NMI interrupt. This line, therefore, does not need to be held high. The external interrupts are all active low at the processor, these interrupt lines are all inverted to active high at the connection end. This allows EMAC to maintain compatibility between EPAC systems.

MASKING INTERRUPTS

The MTBASIC statements used for enabling and disabling interrupts are SETINTS, STORMASK, INTOFF and INTON. INTOFF disables all vectorable interrupts (not including NMI), no matter whether a SETINTS statement has enabled or disabled them. If the INTON statement has been executed, then the interrupts that were set in the last SETINTS statement will become enabled. The STORMASK statement allows you to select which interrupts you want to be enabled after EXITing an interrupt service task, because MTBASIC automatically performs a SETINTS 0 statement which disables all interrupts. You can read the value that was stored by the last SETINTS statement by using the INTMASK function. The data sent to the SETINTS and STORMASK statements and returned from the INTMASK function is in the following format:

bit #	**name**	**bit#**	**name**
0	int0	7	COM 1 Tx
1	int1	8	COM 1 Rx
2	int2	9	COM 0 Tx
3	timer0	10	COM 0 Rx
4	timer1	(11-14 not used)	
5	dma0	15	CSI/O
6	dma1		

Placing a 1 in bit positions 0 - 10 will enable the corresponding interrupt.

EXAMPLES:
```
      SETINTS %111.1111.1111  - WILL ENABLE ALL VECTORABLE INTERRUPTS

      SETINTS %1000           - WILL ENABLE THE TIMER0 INTERRUPT ONLY

      SETINTS %11000          - WILL ENABLE BOTH TIMER0 AND TIMER1 INTERRUPTS
```

VECTORING INTERRUPTS

Upon power up, MTBASIC links TIMER 0 to the internal multitasking handler, and links COM 0 (ASCI 1) to its internal communications handler. If other interrupts are desired to be linked to a task or if it is desired to redirect the TIMER 0 or COM 0 interrupts to a task instead of their default links, the VECTOR statement must be used. The format of the VECTOR statement is as follows:

VECTOR <int#>,<task#>

where <task#> is the number of the task to which you want to link and <int#> is defined as:

<int#>	INTERRUPT
0	INT0
1	INT1
2	INT2
3	TIMER 0
4	TIMER 1
5	DMA channel 0 (DMA 0)
6	DMA channel 1 (DMA 1)
7	Clocked Serial I/O port (CSIO)
8	COM 1 (ASCI 0)
9	COM 0 (ASCI 1)

To link an interrupt to a task . . .

1. Write a task which will handle the occurrence of an interrupt and will clear the interrupt flag or, in the case of level sensitive interrupts, send data to the source of the interrupt to acknowledge the interrupt and turn it off. Do not use PRINT statements within this task or the program will not work properly. If you must use a PRINT statement, then set a flag within the task and have the main task poll the flag and PRINT the desired message.

2. Within the task, choose which interrupts you want to be enabled after the task is finished by using the STORMASK statement. Also this statement can be executed just once in the main task and the next EXIT statement will enable interrupts according to the value selected by this statement. Remember that MTBASIC performs a SETINTS 0 statement automatically only in interrupt service tasks.

3. In the MTBASIC program, before enabling the interrupt, execute a VECTOR statement with the appropriate interrupt number and the number of the task that was written in step 1.

4. For the internal interrupts, include statements which will set the interrupt source's internal registers (i.e. set the baud rate for ASCI 0 or 1, or load TIMER 0's data registers..) , before enabling the interrupt.

5. Enable the appropriate interrupt, with an INTON and SETINTS statement.

AN EXAMPLE USING THE TIMER 1 INTERRUPT

The program requests the count value for TIMER 1 and then prints a message after 10 interrupts.

NOTE: When running the program, values greater than 500 must be entered for the program to work properly, since TIMER 1 is so fast.

```
10 INTEGER X,DUMMY,CNT
20 X=0
30 ERASE
40 VECTOR 4,1 :! THIS LINKS INTERRUPT 4 (TIMER 1) TO TASK 1
50 INTON :! THIS ENABLES THE ENABLING OF INTERRUPTS BELOW
60 SETINTS 16 :! THIS ENABLES TIMER 1 INTERRUPTS
70 PRINT "ENTER COUNT FOR TIMER 1"
80 INPUT CNT
90 WTIMER 1,7,CNT :! TIMER 1 STARTS COUNTING, BEGINNING AT CNT
100 PRINT "WAITING..";
110 IF X > 9 THEN  PRINT "10 interrupts "; : X = 0
120 GOTO 100 :! THE PROGRAM WILL LOOP UNTIL AN INTERRUPT OCCURS
200 ! TIMER 1 INTERRUPT HANDLER
210 TASK 1
220 DUMMY = INPORT($10):!THESE TWO COMMANDS CLEAR THE ...
230 DUMMY = INPORT($15):! TIMER INTERRUPT FLAG.
240 X=X+1 :! X IS INCREMENTED ON EVERY INTERRUPT
250 STORMASK 16 :! THIS WILL RE-ENABLE TIMER 1 AFTER THE EXIT STATEMENT
260 EXIT :! THIS RESUMES THE INTERRUPTED CODE AND SETS INTERRUPTS
```

MTBASIC uses COM 1 (ASCI 0) for standard default communications. Serial communication can, however, be sent to COM 0 (ASCI 1) if desired. The default serial protocol for both COM 0 and COM 1 is no parity, one stop bit, and 8 data bits. Note that COM 1 is referred to as DEVICE 0 and COM 0 is referred to as DEVICE 5 in MTBASIC.

The default baud rates for COM 0 and COM 1 are easily changed by using the EEPOKE statement. These baud rates are read by the system on power-up/reset so the board must be reset or turned off then turned on before the new EEPOKE value becomes effective. Also, caution must be used when changing the baud rate for COM 1 in EEPROM, because if the terminal you are using does not support the new baud rate, you will lose communication with the board and you will not be able to change the baud rate back to the original value!

Baud rates available for COM 0 and COM 1 are 300, 600, 1200, 2400, 4800, 9600 and 19,200. If your board has a 12.288 Mhz crystal then an additional rate of 38,400 is available. Do not use any baud rate other than the ones specified. Other values will give unpredictable results. If a baud rate is not programmed using SETBAUD (see chapter on statements) or if there is an error detected in the EEPROM, the baud rates will default to 9600.

The default communication port COM1 does not require any form of handshaking to operate, although it makes available handshaking lines (handshake out on pin 4 and handshake in on pin 6). It is left up to the user to provide drivers in order to make use of these lines. Some terminals, however, require handshaking to operate, in this case EMAC recommends the use of a null modem cable (refer to the manual that accompanied your terminal to construct a null modem cable). The recommended cable configurations are shown below. "Handshaking" lines are not required by the MICROPAC 180 but may be necessary for the IBM PC and compatibles used as terminal emulators. If the terminal emulator requires handshaking lines, wire a null modem cable by connecting together the PC's RS-232 handshake lines CTS, DSR, DCD and DTR. The diagrams below show the required connections for PCs and terminals with DB25 connectors and PCs with DB9 connectors.

```
            PC or ADM
            and WYSE
MICROPAC    TERMINALS

HDR3 PINS   DB25 PINS
TX   3 ———  3   RX
RX   5 ———  2   TX
GND  9 ———  7   GND

(optional null-modem connections)
           ┌─  5   CTS
           ├─  6   DSR
           ├─  8   DCD
           └─  20  DTR
```

```
MICROPAC    PC

HDR3 PINS   DB9  PINS
TX   3  ———  2   RX
RX   5  ———  3   TX
GND  9  ———  5   GND

(optional null-modem connections)
           ┌─  8   CTS
           ├─  6   DSR
           ├─  1   DCD
           └─  4   DTR
```

Note: When a data terminal has 2 connectors, use the one marked "MODEM".

Since the RS232 voltages (V+ and V-) are generated on board, a single rail power supply (8VDc - 15Vdc) is all that is needed. 12Vdc is recommended whenever possible.

The transmitters and receivers for COM 0 and COM 1, have the capability to be interrupt driven. See the section on interrupts in this manual and the section containing information on the Asynchronous Serial Communication Interface in the MICROPAC 180 Hardware Reference Manual.

Serial Port COM 0 (ASCI 1) can be optionally ordered with a RS422/485 interface replacing the standard RS232 interface. The RS422/485 interface is a balanced line type allowing communications at high baud rates at distances of several thousand feet. In addition this interface has multi-drop capability allowing up to 32 systems to share the same communication lines.

USING COM1 HANDSHAKE LINES

To read the COM1 handshake input from pin 6 of HDR3 examine bit 7 of control port D. This is shown in the following example:

```
10 INTEGER X
20 X = BAND(INPORT (#PPORTD),%1000.0000)
30 PRINT "Handshake in is ";
40  IF X THEN PRINT "ON": GOTO 60
50 PRINT "OFF"
60 GOTO 20
```

To control the handshake out line (pin 4 of HDR3) , set or reset bit 7 of control port D and its RAM shadow as in the following example:

```
5 INTEGER X
10 GOSUB 1000 ' HANDSHAKE OFF
20 PRINT "OFF"
30 WAIT 60
40 GOSUB 1500 ' HANDSHAKE ON
50 PRINT "ON"
60 WAIT 60
70 GOTO 10

1000 INTOFF ' change port and RAM shadow while interrupts are off
1010 X=BAND(%0111.1111,PEEK(#SPORTD))
1020 GOTO 1600

1500 INTOFF
1510 X=BOR(%1000.0000,PEEK(#SPORTD))
1600 OUTPORT(#PPORTD),X ' write the new value
1610 POKE(#SPORTD),X ' write back to the RAM shadow
1620 INTON
1630 RETURN
```

Chapter 15: Input and Output Examples

This chapter groups together all the MTBASIC instructions, with example programs, which are needed to use the input and output devices (serial communication is discussed in another chapter).

INPORT (<expr>)

INTEGER

Reads input port <expr> returning an 8 bit value.

Example:

```
        10 INTEGER P
        20 PRINT "Which port (0-65535)?"
        30 INPUT P
        40 PRINT "I/O port # ";P;" holds a ";INPORT(P)
        50 GOTO 20
```

ADCIN(<expr>)

INTEGER

This returns the digital value of the analog signal applied to the A/D convertor channel selected by <expr>. The value returned by channels 0 to 7 will be in the range 0 to 1023 for the 10 bit A/D convertor. If the 12 bit A/D option is installed, channels 8 to 15 will return a value in the range of 0 to 4095. Channels 0 to 7 can only be used with the 10 bit A/D and channels 8 to 15 can only be used with the 12 bit A/D option. Channels 0 to 7, or 8 to 15 correspond to the odd numbered pins 1 to 15 of HDR7.

Example:

```
        10 INTEGER CHANNEL
        20 FOR CHANNEL = 0 TO 7
        30 PRINT "CHANNEL ";CHANNEL;" = ";ADCIN(CHANNEL)
        40 NEXT CHANNEL
        50 WAIT 80
        60 ERASE
        70 GOTO 20
```

DACOUT <expr1>,<expr2>

DACOUT sends the value of <expr2>, which must range from 0 to 4095, to the digital to analog convertor channel (0 to 3) selected by <expr1>. This statement requires the digital to analog convertor option (part number E301-04).

Example:

```
10 REM THIS PROGRAM PRODUCES A REPETITIVE ASCENDING RAMP SIGNAL ON
20 REM CHANNEL 0 WHILE PRODUCING A DESCENDING RAMP SIGNAL ON CHANNEL 1.
40 INTEGER X
50 FOR X=0 TO 4095
80 DACOUT 0,X
90 DACOUT 1,4095-X
100 NEXT X
110 GOTO 50 : REM PRODUCE THE NEXT RAMP
```

OUTPORT <expr 1>, <expr 2>

The OUTPORT statement sends the byte <expr 2> to output port <expr1>. No check is made to see if anything is connected to the port. OUTPORT sends only the lower 8 bits of <expr 2> to the port. The following program sends the numbers FF hex to 0 to the high current driver port. This port is accessible through the odd numbered pins 1-15 of HDR6.

```
1  INTEGER X
5  PPICFG %1011 ' SET UP PORTB FOR OUTPUT, THE REST ARE INPUTS
10  FOR X=$FF TO 0 STEP  - 1
20  OUTPORT (#PPORTB),X
30  WAIT 1
40  NEXT X
```

PPICFG <expr>

This configures the directions for PPI ports A, B and the lower and upper 4 bits of port C based on the value of <expr> which is formatted as follows:

bit	
0	port C upper 4 bits (1 = input, 0 = output)
1	port C lower 4 bits (1 = input, 0 = output)
2	port B (1 = input, 0 = output)
3	port A (1 = input, 0 = output)

The program below makes the upper half of port C into inputs and the lower half of the port C and port B into outputs (remember that if you want to detect the level of the port B output using a scope or meter, pullup resisters should be used) . If the binary value that is applied to the upper half of port C is less than 12 then the OUTPUT BIT (see diagram below) corresponding the the input value will be set to 1, with all other OUTPUT BITs reset to 0. For example, if the binary value being input the the upper half of port C is 8 then OUTPUT BIT 8 will be 1. This program simulates a 4 to 12 decoder.

```
PORT C (lower)        PORT B
  3  2  1  0     7  6  5  4  3  2  1  0   PORT BIT NUMBER

 11 10  9  8     7  6  5  4  3  2  1  0   OUTPUT BIT NUMBER
```

EXAMPLE

```
10 INTEGER DATIN
20 PPICFG %0001 : REM UPPER OF PORT C IS INPUT, THE REST ARE OUTPUTS
30 DATIN = INPORT(#PPORTC): REM Get data from the upper 4 bits of port C
40 DATIN = DATIN / 16: REM Move data from the upper 4 bits to the lower
50 DATIN = EXP(LOG(2)*DATIN): REM Two to the power of DATIN
60 OUTPORT(#PPORTB),DATIN: REM OUTPORT always sends only the lower 8 bits
70 REM Output the upper 8 bits to port C.  Since only the lower 4 bits of
80 REM  port C are output, output will only go to the lower 4 bits
90 OUTPORT(#PPORTC),DATIN / $FF
100 GOTO 50
```

AUTOSTART PROGRAMS

If an EPROM programmer is not accessible or frequent changes to an application are anticipated, the application can be stored on RAMDISK and automatically loaded whenever the board is turned on. This is done by simply writing the application program and then storing it on RAMDISK under the name "AUTOSTART.BAS". When the board is turned on and MTBASIC determines that there is not an application EPROM in memory slot 1, it will search the directory for "AUTOSTART.BAS" ,load it (which means you can embed direct commands within the file, such as NOERR), and execute it.

If you later decide that you don't want to autostart this program, load the program and save it under a different name, then delete "AUTOSTART.BAS". If you embed a NOERR direct command in "AUTOSTART.BAS" and you don't have an option within the program for stopping it, you can stop the program by bringing the NMI interrupt pin high (in connector ST-1). Then you can save the program under a different name and delete "AUTOSTART.BAS".

EPROM BASED PROGRAMS

When you have worked all the bugs out of your MTBASIC program and you desire to put a permanent version of the program into EPROM, you have two options:

Option #1: Using EMAC's EPROM Programmer.

EMAC's EPROM programmer (E020-8) allows you to program EPROMs of a variety of types and voltages. The smallest EPROM supported is a 2764 (8K x 8) and largest EPROM supported is 27512 (64K x 8). Each EPROM type is given an identification number. The 6 EPROM type numbers supported are as follows:

```
TYPE #1     27512 ( 64K x 8 ) EPROM WHICH PROGRAMS AT 12.5 VOLTS.
TYPE #2     27256 ( 32K x 8 ) EPROM WHICH PROGRAMS AT 12.5 VOLTS.
TYPE #3     27128 ( 16K x 8 ) EPROM WHICH PROGRAMS AT 12.5 VOLTS.
TYPE #4     27128 ( 16K x 8 ) EPROM WHICH PROGRAMS AT 21.0 VOLTS.
TYPE #5     2764  (  8K x 8 ) EPROM WHICH PROGRAMS AT 12.5 VOLTS.
TYPE #6     2764  (  8K x 8 ) EPROM WHICH PROGRAMS AT 21.0 VOLTS.
```

MTBASIC has four direct commands which support EMAC's EPROM programmer

BURN <source>,<dest>,<qty>,<type>

This copies <qty> bytes from system address <source> to EPROM address <dest> for the EPROM indicated by <type>. The value for <source> must be greater than $8000.

ERASECHK <addr>,<type>

ERASECHK tells whether the bytes from 0 to <addr> are erased. The value <type> indicates what kind of EPROM you are checking.

TBURN

TBURN first checks to see if the EPROM in the programmer is blank and then it allows you to take a program that was RCOMPILEd and copy it to a 27512 EPROM.

VERIFY <source>,<dest>,<qty>,<type>

VERIFY tells whether the <qty> number of bytes at <source> match the <qty> number of bytes at <dest>, for the EPROM described by <type>. The command will terminate and give the source address followed by the value at that address, and the destination address followed by the value at that address, if there are differing values.

NOTE: To insure desired results:

1. When inserting or removing EPROMS, make sure the EPROM burner power LED is off.

2. When performing an ERASECHK, VERIFY or BURN command make sure the EPROM type was correctly entered.

To make a permanent version of a program using EMAC's EPROM programmer, do the following:

1) Make sure the program is fully debugged. If it is desired to make a NOERR version of the program (see NOERR in the chapter on direct commands), and the watchdog timer is enabled by jumpering pins 2 and 3 of J5, be sure to put the WATCHDOG statement throughout the program to keep the watchdog timer from timing out and resetting the board. Fully test the NOERR version of the program, and save the working version to RAMDISK.

2) At the MTBASIC prompt, type "RCOMPILE" and enter. This will compile your program so that it will execute at a different location in memory.

3) Insert a type 1 (27512) EPROM into the EPROM programmer, then at the MTBASIC prompt type "TBURN" and enter. At this point MTBASIC will give a message if the EPROM has not been erased. If it is erased, "Burning..." will be displayed and below it will be the number of bytes left to program (in hex format). If it burns all the bytes without an error, "Burnt" will be displayed and the MTBASIC prompt will appear.

4) Once the EPROM programmer power indicator is off, remove the EPROM. Turn off the power to the board and carefully remove the MTBASIC EPROM. Insert the EPROM that was just programmed, being careful not to put the chip in backwards then power up the board. The program should start running.

Option #2: Using an EPROM Programmer That is Connected to a PC.

It is possible for MTBASIC to generate an Intel hex file of your program so that you can make an EPROM using another computer and EPROM programmer. You can send an Intel hex file to an IBM PC through EMAC's terminal emulator package ECOM, or PROCOMM. The MTBASIC direct command that generates an Intel hex file is RHEX. The RHEX command sends to the console an Intel hex file which contains all of the necessary MTBASIC routines linked with the program that was compiled using the RCOMPILE command. Perform the following steps:

1) Connect the board to the PC and make sure you have full communication through the terminal emulator package.

2) Make sure your MTBASIC program is fully debugged. If it is desired to make a NOERR version of the program (see NOERR in the chapter on direct commands), and the watchdog timer is enabled by jumpering pins 2 and 3 of J5, be sure to put the WATCHDOG statement throughout the program to keep the watchdog timer from timing out and resetting the board. Fully test the NOERR version of the program, and save the working version to RAMDISK.

3) At the MTBASIC prompt, type "RCOMPILE" and enter. This will compile your program so that it will execute at a different location in memory.

4) If using ECOM, go to step 5, otherwise at the MTBASIC prompt, type "RHEX" but do not press enter.

5) Choose the download option from the terminal emulator. The file that will be downloaded will be an ASCII file and its filename should have a ".hex" extension. Now press enter and the Intel hex file should start loading into the PC.

6) After the file is downloaded you can exit the terminal emulator package and burn a 27512 EEPROM using the Intel hex file that was just downloaded.

7) Turn off the power to the board and carefully remove the MTBASIC EPROM. Insert the EPROM that was just programmed, being careful not to put the chip in backwards, then power up the EPAC. The program should start running.

32K RAM APPLICATIONS

Some programs, if they don't use too much variable space, may be compiled to run on boards that have only 32K of RAM (they must be compiled on boards with 128k of RAM though). This allows you to save money by using less expensive 32k RAMs, which is an important consideration when you need to populate many boards with RAMs. To do this, use "RCOMPILE A" in place of "RCOMPILE" in the above steps.

Chapter 17: Troubleshooting

If you don't receive the MTBASIC prompt immediately, a check of the following items should aid you in determining the source(s) of trouble.

POWER

Check to make sure the power switch from your power supply is in the on position. This power supply should be from 8 to 15 volts DC. The EPAC's 3-contact power strip terminal ST2 should be wired as follows:

Positive	screw terminal # 1
Ground	screw terminal # 2

COM1 SERIAL COMMUNICATIONS

See the section GETTING STARTED in this manual for proper connection. Consult the user's manual of your host terminal or computer to insure proper wiring of the RS232 line (TXD, RXD etc.).

BAUD RATE

Make sure that the baud rates match. The default baud rate and protocol is 9600 baud, 8 data bits, 1 stop bit and no parity. See the chapter on serial communication.

RUNNING PROGRAM

The board may be executing a program when communication is attempted. A power-down --- power-up sequence should yield the MTBASIC logon message.

MISSING WATCHDOG TIMER JUMPER

Be sure that the watchdog jumper is installed (see your hardware manual).

JAMMED UART

For some reason one of the UARTs may be jammed. Hitting the RETURN key might free the troublesome UART. If not, a power-down / power-up may be required.

MISSING IC

The MTBASIC EPROM or an application EPROM must reside in the 1st slot of the board and an 128K x 8 RAM (or 512K) in the 2nd slot.

**** NOTE **** IF THE APPLICATION IS NOT DESIGNED TO PRODUCE A PROMPT OR ALLOW COMMUNICATIONS, THEY WON'T OCCUR.

AUTOLOAD

Check the directory of the RAMDISK. The presence or absence of the file "AUTOSTART.BAS" will determine if it is going to run or not (see the chapter on stand alone applications).

Appendix: Error Messages

***** BAD INPUT. PLEASE RE-ENTER ***** : displayed at runtime if the data entered in response to an INPUT statement doesn't match the arguments given. Just retype the input data properly.

DATA STATEMENT DOES NOT MATCH READ: occurs when a READ or RESTORE is encountered, but no DATA statements have been found. All DATA statements must be before the first READ.

EXPRESSION ERROR: occurs when a mathematical expression has been formatted incorrectly.

FILE NOT FOUND: is displayed when a LOAD command is issued to a non-existent file.

FUNCTION ERROR: occurs when a program is compiled if the function had an argument in the wrong mode, or if the wrong number of arguments were given for the function. At runtime, a FUNCTION ERROR may appear if the argument to a function is out of range, for example, if the square root of a negative number is taken.

ILLEGAL DIRECT COMMAND: occurs if an unknown direct command is entered.

ILLEGAL DEVICE NUMBER: occurs if a device number is specified which is not in the range specified by the DEVICE statement description.

ILLEGAL PRINT FORMAT: is displayed at runtime if the format given in an FPRINT statement does not match the number of arguments in the FPRINT statement (if there are more things to be printed than there are formats), or if the format specified is not correct.

ILLEGAL TRANSFER ERROR: the error returns line numbers as follows:

<line#> → <train#> Illegal Transfer Error

This indicates that a transfer (i.e. GOTO or GOSUB) was made to a line outside of the current task. The line number being transferred to is <train#> and <line#> is the line number where the error occurred. If you receive the message without the line numbers, the error occurred before the bank compiling messages were enabled. Use the "B" option when compiling to find the line number(s) of the transfer error(s).

IMPROPER DATA TO INPUT STATEMENT: occurs when data is being read from a file and the data doesn't match the arguments of the INPUT statement.

INSUFFICIENT DISK SPACE: occurs during a SAVE operation if the RAMDISK is full or nearly full.

LINE NUMBER DOES NOT EXIST: occurs if a line number is referenced in a GO TO, GOSUB, or IF statement and cannot be found.

LINE NUMBER ERROR: occurs when an illegal line number is used. Line numbers must be integers from 1 to 32,767.

MISUSE OF STRING EXPRESSION: occurs when a string expression is used in a place where a real or integer expression is needed, or if a real or integer expression is used in lieu of a string.

NO COMPILED CODE: is displayed if a GO command is entered, but no RUN or COMPILE has taken place or STATUS is issued, but the program hasn't been compiled yet.

NOT ENOUGH MEMORY TO COMPILE PROGRAM: may occur when a very large program is compiled. The size of the program can be reduced by shrinking arrays, removing REMs, etc. The following occurs when a task (even the lead task) is too big.

Task <task#>
<line#> Not Enough Memory to Compile Program

You will need to break down the task specified by <task#> into one or more other tasks or change portions of the task to functions and put them in the lead task (task 0), if there is room. The error will also occur if the compiler runs out of banks, but in this case **Task <task#>** is not shown.

QUOTE OR PARENTHESIS MISMATCH: occurs when a program is being typed into MTBASIC and an odd number of double quotes (") or parentheses are found.

RETURN WITHOUT GOSUB: occurs during program execution if a RETURN is encountered when no GOSUB is active. All GOSUBs must have a corresponding RETURN.

STATEMENT FORMED POORLY: occurs when MTBASIC can't quite figure out what the entered statement is supposed to be.

STATEMENT ORDERING ERROR: occurs if an INTEGER, REAL, or STRING statement appears after executable statements in the program.

STRING LENGTH EXCEEDED: occurs when a string exceeds 127 characters or a string variable exceeds the maximum size assigned to it in the STRING statement.

STRING SPACE EXCEEDED: occurs if one line of the program requires too many string temporaries to evaluate. Try splitting the line into several simpler ones.

STRING VARIABLE ERROR: displayed when a variable is used incorrectly, for example, if a string is used as a real or integer.

SUBSCRIPT OUT OF RANGE: occurs if the subscript of a dimensioned variable exceeds the range assigned in an INTEGER or REAL statement.

TASK ERROR: can be caused by any of the following:

1) receiving a hardware interrupt which was vectored to a non-existent task.
2) Trying to START a non-existent task
3) Trying to START TASK 0
4) Trying to use more than 32 TASK statements.

TOO MANY VARIABLES: occurs if more than 255 variables are used. If you need more than 255 variables try using arrays.

UNDEFINED VARIABLE: occurs during compilation if a variable is encountered which was not defined in an INTEGER, REAL or STRING statement.

UNMATCHED FOR...NEXT PAIR: occurs during execution of a program if a NEXT is found without a FOR.

UNRECOGNIZABLE STATEMENT: is displayed when typing in a program and MTBASIC can't quite figure out what the statement is supposed to be.