MTBASIC

Multitasking BASIC Compiler

SOFTAID, Inc.

P.O. Box 2412 Columbia, MD 21045-1412 301/792-8096

MTBASIC

MULTITASKING BASIC COMPILER

Softaid, Inc.

Columbia, MD 21045

Revision 09 (9/85)

Table of Contents

Part II - User Reference

Warranty

Softaid, Inc. warrants the media on which MTBASIC is delivered against physical defects for 30 days after the date of purchase. In the event of destruction of the media during the warranty period, send the original disk to Softaid and the media and software will be replaced for a $3.50 shipping and handling charge.

Softaid, Inc. will not be held responsible for any damages caused by use or misuse of MTBASIC. Softaid, Inc. provides MTBASIC with the understanding that the user will determine its fitness for any particular application and Softaid specifically disclaims any implied warranties.

Software License Agreement

This software product, MTBASIC disk and manual, is copyrighted and all rights are reserved by Softaid,Inc. The distribution and sale of this product are intended for the use ot the original purchaser only and for use only on the computer system specified. Copying, duplicating, selling or otherwise distributing this product without the express written permission of Softaid, Inc. is a violation of U.S. Copyright Law and hereby expressly forbidden.

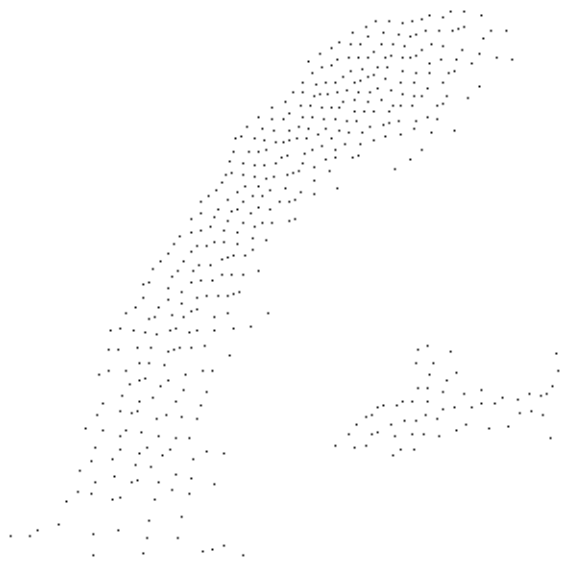
Object code generated by MTBASIC may be distributed without consulting Softaid, Inc. No royalty fee for the distribution of the object code is required, but credit must be given to Softaid, Inc. in the documentation and code of any program used for distribution (any compiled programs given or sold to anyone other than the Purchaser). The credit must consist of a line of the following general content:

Compiled using MTBASIC, by Softaid, Inc., P.O. Box 2412, Columbia, MD 21045.

Update Policy

MTBASIC is periodically updated to remove bugs and enhance its operation. Purchasers of MTBASIC can obtain updates by sending the original MTBASIC disk and $20.00 to:

Softaid, Inc.
P.O. Box 2412
Columbia, MD 21045



PART ONE

Using MTBASIC

Chapter One: Introduction to MTBASIC

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

Softaid strongly encourages the user to read this entire manual before attempting to run the compiler. MTBASIC is an extremely powerful system. It provides commands for controlling real time multitasking processes. These commands are not standard Basic - indeed no other Basic has these features. A thorough reading of the manual will prevent problems from occurring.

MTBASIC requires a minimum of 128K of memory on PC-DOS systems. CP/M versions require at least 48K of available memory.

No runtime license is required to distribute programs compiled by MTBASIC. Per the software license agreement, the manufacturer of software products compiled by MTBASIC must credit Softaid in the product's documentation.

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 takes many 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 a disk file 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. Many people will enjoy writing games using MTBASIC's multitasking features, since these features can give games a degree of realism never before possible.

MTBASIC also supports windowing. The user can partition the CRT 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 3) before windowing commands are used.

Finally, it is important to note the constraints under which MTBASIC operates. Softaid 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, must keep the entire compiler, the runtime package, the entered source code, and the generated object code all resident during the debugging process. This severely restricts the amount of memory which is available for the user. In order to keep a large amount of user memory available, Softaid has elected to reduce the command set to the minimum

necessary.

GETTING STARTED

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

1. Begin by copying the MTBASIC master disk to a work disk. Use the commands and copy program which are part of your operating system. Put the master disk away and save it to make more back ups. Also, if you call Softaid for help, we will ask for the serial number of the original disk. If you order an MTBASIC update, you will need to return the original disk.

2. Before using windowing or the example programs on the disk, you must first configure MTBASIC for your particular terminal (see Chapter 3). If you want to begin with the example given in this section, you do not need to configure your terminal first.

3. To begin, first get into the compiler. Your responses will be in **bold** face, the computer's responses will be in plain face.

```
A>MTBASIC

MTBASIC Multitasking Basic Compiler Version 2.5
Copyright 1984, Softaid, Inc., Columbia, MD

>
```

4. 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.

```
>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$
```

5. This is the example program. The next series of commands 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
```

6. There is an error in line 50. The second comma simulates a tab after "Hello,". The next step is to correct this. To edit line 50, enter the line number and the new line.

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

7. This time, for example, 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
```

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

```
> SAVE PROG
>
```

After you finish and leave MTBASIC (see step 10), if you want to run

PROG again, re-enter MTBASIC (see step 3) and enter:

```
› LOAD PROG
›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$
›END
```

MTBASIC will list the program as it is loaded. Note that MTBASIC added an END statement when the SAVE was done. To run this program now, see step 5.

9. If you want to save the example program as a stand alone program (one that will run independently of MTBASIC), use the DISK COMPILE command.

```
›DISK COMPILE PROG
COMPILED

Start address      : 0100
End address        : 304F
Variable start     : C900
›
```

10. To run PROG.COM, exit MTBASIC (return to the operating system) and then run PROG as though it were any other .COM program, as follows:

```
›BYE

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

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.

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.

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).

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 8; however, no more than 255 variables may be declared.

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.

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

Chapter Two: 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.

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

On some MS-DOS systems, the interrupt vector for the system's real time clock must be installed before multitasking can take place. The TICINT direct command is used to do this, and is fully documented in Chapter 7 of this manual.

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 the computer is so fast.

The interrupt, then, is the basic unit of task switching. Within MTBASIC, interrupts 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.

CP/M systems come in many configurations. Some have a source of hardware interrupts that can be used as tics, while others do not. So multitasking may be used in all CP/M systems, Softaid has provided a method of simulating interrupts (called "software interrupts"). MS-DOS systems do not use software interrupts, since all MS-DOS systems are equipped with a hardware tic source. The direct command TICS ON makes MTBASIC generate software interrupts in CP/M systems, while in MS-DOS systems TICS ON enables the hardware tic source.

Under CP/M, the use of hardware interrupts requires the programmer to write an interrupt service routine to process the hardware interrupts (every source of hardware interrupts will interface to the computer differently). 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.

Software interrupts, which MTBASIC generates automatically if TICS ON is specified, are much easier to use because no special hardware is required, so no special service routine is needed. Software interrupts come at a rate proportional to the amount of time it takes to execute each individual statement within the program, which varies considerably. Precise timing with software interrupts therefore is not possible, but in most applications this is not really important. PC-DOS users need not be concerned with software interrupts since MTBASIC always uses the PC's built in clock to sequence tasks. Issuing TICS ON under PC-DOS enables the clock.

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 disk controllers generate an interrupt when data is ready to be read from the disk. MTBASIC allows the user to process this interrupt and start a task whenever a device interrupt occurs. Using this concept, a disk 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 and JVECTOR instructions, described in Chapter 8, are 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 RUN 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 RUN statement, the schedule interval, is not significant, since the tasks never EXIT.

```
20 RUN 1,100
30 RUN 2,100
40 PRINT 0
50 GO TO 40
60 TASK 1
70 PRINT 1
80 GO TO 70
90 TASK 2
100 PRINT 2
110 GO TO 100
```

A more sophisticated version of the previous program is shown below. Note that in this program there are no GO TO statements, other than the one in the lead task. Each of the two subtasks 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 RUN statement, the tasks will be "reborn" after the number of tics specified in the RUN 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 RUN 1,100
30 RUN 2,50
```

```
40 GO TO 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 RUN 1,I
50 IF I<>100 THEN 50
60 PRINT "100 tics counted"
70 I=0
80 GO TO 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.

Thousands of applications for multitasking games exist. A video game can be programmed in one task, and another task can be scheduled to run every few seconds to make aliens appear on the screen. Or, if two terminals exist and MTBASIC's device I/O commands are used, two players can play a game against each other without waiting turns to enter data, with INPUT statements active at each terminal simultaneously.

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 GO TO 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 RUN statement must be executed for that task.

If a hardware tic source does not exist, then for a multitasking program to execute, the TICS ON direct command must be entered before the program is compiled. This will cause compilation of software interrupts into the program. TICS ON must *always* be issued on MS-DOS systems to enable MTBASIC's multitasking code.

Here is a listing of the exact sequence of inputs typed by a programmer to enter and run a simple multitasking program:

```
20 RUN 1,10
30 PRINT "LEAD TASK"
40 WAIT 100
50 GO TO 30
60 TASK1
70 PRINT "TASK 1"
80 EXIT

TICS ON
RUN
```

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. 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. On an IBM-PC, the system clock issues a tic about 18 times/second.

The INTOFF and INTON statements provide convenient methods of turning multitasking on and off within a program. At any point in a program INTOFF may be executed to stop multitasking. The currently

executing task will remain the only task which executes until an INTON is executed. This gives the user the ability to implement a task priority system. High priority tasks should execute an INTOFF to allow them to run to completion, so other tasks will not take processor time away from the high priority tasks. When the high prioritiy task is finished, it must execute an INTON so that the other tasks can continue.

VECTOR and JVECTOR create links to MTBASIC tasks. Once created, these links cannot be removed. If a link is erroneously created and must be removed, either reboot or restore the contents of the location specified in the VECTOR/JVECTOR statement.

Device handlers using hardware interrupts must not re-enable interrupts. Interrupts will be automatically re-enabled when the EXIT statement is executed. Additionally, device handlers must not use WAIT statements, since WAIT re-enables interrupts. Device handlers must run to completion rapidly. If extensive processing is needed, the device handler should RUN an additional task to do the processing. Device handlers should not do I/O via PRINT, INPUT, or FPRINT statements, since these will re-enable interrupts.

As described in Chapter 8, the RUN statement starts execution of the indicated task one tic after the RUN 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 incarnation of the task. For example, the following program prints a message 200 tics after the RUN statement for task 1 is executed.

```
10 INTEGER I
20 I=0
30 RUN 1,199
40 GO TO 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 RUN 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 RUN 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 RUN statment for that task is executed. CANCEL does not kill the task immediately. The task will continue executing until an EXIT statement is encountered. 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 never gets a TASK statement.

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.

MTBASIC starts with interrupts enabled, therefore, DO NOT turn on your hardware interrupts until the links have been established by executing a VECTOR/JVECTOR. Under CP/M the proper INTMODE must also be set.

On CP/M systems, DO NOT set an INTMODE different than the one needed by your system's BIOS if your system uses hardware interrupts! If hardware interrupts are being used, it may be necessary to disable them around disk I/O statements, unless your system requires interrupts for I/O. Any interrupt processing degrades throughout.

The IBM-PC supports only one interrupt mode, so the INTMODE instruction is not used. Similarly, the JVECTOR statement is not needed.

Hardware interrupts (including TICS ON on the IBM-PC), are ignored during BIOS calls, since neither CP/M nor MS-DOS is re-entrant. Therefore, if your program does a lot of I/O, interrupts may occasionally be missed.

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.

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

Type this program in and RUN it. It prints "MAIN PROGRAM" lots of times without stopping. That could have been done with a simple two-line program. So what about multitasking? Type TICS ON. This tells the compiler to generate code that simulates interrupts. Type RUN again and watch the results. 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 TICS ON command turns on "tics" to be used like a clock by the program. When the TICS ON command was issued before RUNning the program a second time, it told the compiler to add some code to the finished program that would increment a counter like a real time clock. Each time the computer increments the counter it creates one tic. The tics cause the computer to switch between tasks in the program. On MS-DOS systems, TICS ON turns on MTBASIC's multitasking code, so the computer's clock can cause real interrupts.

With interrupts on, the RUN statement tells the program to execute the task indicated by the first number (1) when the clock count is evenly divisible by the second number (100). Variables can be used for the second number in order to permit changing the scheduling interval for the task. Do not confuse the RUN statement, which starts a task going, with the direct command RUN, whcih starts a program running.

At line 30 the program is set to run a task, but which task? The "1" following RUN is the task number (in this case there is only one possible task, but there could be as many as ten). 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 RUN 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 Three: 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 Chapter 9, are:

ERASE - erases the CRT.

CURSOR - repositions the cursor within a window.

WLCEAR - 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.

CP/M INSTALLATION

You must configure MTBASIC for your particular terminal before using windowing since windowing makes use of several terminal dependent codes which must be defined. On CP/M systems, MTBASIC comes pre-installed in the ADM-3A (identical to Kaypro) format. The INSTALL program on your MTBASIC disk will do the installation. To use INSTALL:

Backup the disk supplied by Softaid and put the master away in a safe place.

Using the backup disk, put the disk with INSTALL and MTBASIC (they must be on the same disk) in your system and press <control-C>.

Type the drive number (of the drive this disk is on - i.e, set this disk to be the default drive), colon.

Enter MTBASIC and issue a LOAD INSTALL command. When the program is LOADed, type RUN and follow the instructions given by the program.

When INSTALL is finished, your MTBASIC.COM file will contain a configured MTBASIC. Don't worry about making an error; you can re-install at any time.

Once MTBASIC is installed, 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).

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" (defined by you in INSTALL, usually an ASCII escape) followed by exactly one character.

MS-DOS INSTALLATION

MTBASIC comes pre-installed for PC-DOS and MS-DOS systems. However, MTBASIC requires that the ANSI device driver be installed. The DOS disk supplied with your system will have the file ANSI.SYS on it. Copy this file to your boot disk. Next, create the file CONFIG.SYS on your boot disk. CONFIG.SYS is the system configuration file and is not needed by MS-DOS unless device drivers are to be installed (as in this case). Your MTBASIC disk has a CONFIG.SYS file which can be copied to the boot disk.

To use windowing on MTBASIC, CONFIG.SYS must contain at least the following line:

```
DEVICE = ANSI.SYS
```

Whenever the computer is booted, this file must be on the boot disk. After completing these steps, reboot the computer and MTBASIC is ready to run.

DEMONSTRATION PROGRAMS

The MTBASIC disk you received contains several programs demonstrating various types of windowing. Try running these programs to gain some confidence with both windowing and multitasking.

WIND1.BAS - contains 9 separate tasks, each of which is assigned to a window. Each task writes its name to its window at a different rate. To run it, type:

```
LOAD WIND1.BAS
```

```
TICS ON
RUN
```

On some MS-DOS systems, the real time clock interrupt vector is not the same as the one MTBASIC is configured for. You may have to issue the TICINT command (chapter 7) to run WIND1.BAS.

WIND2.BAS - is a demonstration of pop down menus. This is a simulation of a word processing application. To run it, type:

```
LOAD WIND2.BAS
RUN
```

WIND3.BAS - generates random windows on the CRT. To run it, type:

```
LOAD WIND3.BAS
RUN
```

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.

(CP/M systems only - When running a multitasking program using software tics with an INPUT or INPUT$ active in one task, avoid long statements in other tasks. When using software tics, MTBASIC does context switching at the completion of each MTBASIC line. A long line of code can cause the input to lose characters if someone is typing fast. For example, avoid constructs like FOR K=1 TO 10000: NEXT K. Break these statements up into two lines.)

Framing is a very useful, but easily misunderstood feature. Whenever WFRAME is executed, an outline is drawn around 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 Four: 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 recursion - recursive programs 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. Softaid has never had a problem with programs that call themselves ten times and we have successfully written programs that call 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 destroying the compiler. If you need a greater depth (use a backup disk), 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.

Chapter Five: File and Device I/O

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/or store data to a disk 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 a game where several terminals may be in use at the same time. 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 the "SAVE" and "LOAD" commands which allow the programmer to store and retrieve programs on disk. These commands are not discussed here (see Chapter 7) and cannot be part of an actual MTBASIC program. SAVEd files are stored in ASCII and can therefore be altered (or generated) by an editor.

Detailed descriptions of the file handling statements are given in Chapter 8, but their functions will be summarized here. The statements which are related to file and device handling are: OPEN, CLOSE, FILE, FIELD, PUT, RGET, SEEK, DELETE, PRINT, INPUT (or INPUT$), and FPRINT. The GET, CVI, CVS, MKI$, MKS$, ERR and ERR$ functions, described in Chapter 10, are also associated with file handling.

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

Before a 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 "file number" with the file name. That file number is then used within the program to reference that file. For example, if the

statement OPEN 0,1,"ABC" is executed, then file 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 file number from the file name, making that number available for use with other files. Also, in the case of a write file, the last record processed is written to the file. Write operations take place to a buffer in memory which is written to disk only when the buffer becomes full. Therefore, after the last file write is done, data may still reside in the memory buffer and may not yet be written to the file. Executing a CLOSE guarantees that all information will be written to the file.

Once a file has been opened, it is only necessary to execute a FILE statement to direct I/O to that file. This tells the system that all subsequent I/O will take place to the file number specified in the argument of the FILE statement. For example, executing FILE 1 causes all inputs and outputs to be redirected to the file previously associated with FILE 1 by an OPEN statement. A user may open up to three files and direct I/O between these files simply by changing the currently active file number using FILE statements.

The ERR and ERR$ functions can be used to determine if errors occur during file I/O. These functions are only significant during disk file I/O, since I/O performed to other devices does not create errors. ERR returns the error number of the error encountered during a file read or write, and ERR$ returns a string containing the error message. It is strongly recommended that whenever a file 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 10.

Some programs may need to read binary files. MTBASIC provides the GET function to accomplish this. GET reads a single character from the currently selected file and returns an integer version of the character.

EXAMPLE

The following is a listing of an MTBASIC program which demonstrates the use of file I/O. The program copies one file to another. Note that if a

program needs to copy files containing commas, the INPUT$ statement must be used because in INPUT statements, commas delimit the end of an input string.

```
10 STRING A$(80), B$(30)
20 OPEN 0,1,"INFILE"
30 OPEN 1,2,"OUTFILE"
40 FILE 1
50 INPUT A$
60 IF ERR=10 THEN 100: ! Branch if end of file
70 FILE 2
80 PRINT A$
90 GO TO 40
100 B$=ERR$
110 FILE 0
120 PRINT ERR$
130 CLOSE 1
140 CLOSE 2
150 STOP
```

RANDOM FILES

MTBASIC makes no distinction between sequential and random files. By definition, a sequential file is one which is accessed in a serial manner. Every time a read or write is performed to a sequential file, the successive reads or prints operate on the next record in the file. By contrast, a random file is one in which I/O may take place any where in the file. Random files can also give the user complete control over the file's format. Although MTBASIC always operates in a sequential manner, the user may access any particular record by performing a SEEK.

The SEEK statement performs random I/O operations by positioning the system's file pointer to the record specified as its argument. Records are 128 byte long segments of the disk, unless the record size is modified via the FIELD statement. All I/O operations take place in multiples of records. In normal sequential I/O operations the user is not concerned with records, since the I/O is performed in a fashion similar to that of reading or writing to the console.

The FIELD statement can be used to specify the format of a record of

data in a random file. When a file's record structure is defined with FIELD, then PUT and RGET *must* be used to transfer data to the file; normal PRINT and INPUT statements will not work. FIELD defines the record's formal, 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$, and ending with 4 bytes of C$. In this case, SEEK will work in multiples of 9 bytes (1 record).

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, so LSET and RSET instructions are not needed. 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.

Each open file can have a different FIELD statement associated with it, so the FIELD statement must come *after* the FILE statement selecting the file.

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 FILE 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. SEEK may be used to go to any particular record.

For compatibility with other Basics, MTBASIC provides the CVI, CVS, MKI$, and MKS$ functions to convert numeric data to binary strings and vice versa. Since MTBASIC allows integer and real arguments in the FIELD statement, these functions are really not needed. Remember that these functions use binary strings which are not at all readable.

Since CVI converts a string to an integer, and MKI$ does the reverse, then CVI(MKI$(<any integer>))= <the same integer>. The same is true of CVS and MKS$ for real numbers.

USER CONFIGURED I/O DEVICES

The file statement is also used to redirect I/O to alternate devices. As described in Chapter 9, FILE arguments 1, 2, or 3 cause the I/O to be redirected to a file as described above. If the argument is 0, I/O will be directed to the console. An argument of -1 causes the output to go to the printer, and arguments of 4, 5, or 6 cause I/O to take place to user specified alternate devices.

To perform I/O to a user configured device, it is necessary to patch in the machine language driver for that device. The device driver itself can be patched into free memory using POKE statements. In addition, a linkage must be provided from MTBASIC to the device driver.

In CP/M systems, starting at location 106H in MTBASIC, a jump table exists to provide the linkage from MTBASIC to a user configured device driver. In MS-DOS systems, the table is at location CS:106H, where CS: is MTBASIC's loaded code segment address. The jump table takes the following format:

```
JP      0  ;  DEVICE 4   CHARACTER OUT LINK
JP      0  ;  DEVICE 4   CHARACTER IN LINK
JP      0  ;  DEVICE 5   CHARACTER OUT LINK
JP      0  ;  DEVICE 5   CHARACTER IN LINK
JP      0  ;  DEVICE 6   CHARACTER OUT LINK
JP      0  ;  DEVICE 6   CHARACTER IN LINK
```

For each of the three possible devices a jump exists to both the read and write routines for the device. The user need only patch the address of the device driver into the address field of the jump instructions. The POKE statement is useful for this in CP/M systems but not in MS-DOS systems because POKE will operate on MTBASIC's data segment.

The input routine for the device being configured must return a 0 in register A if no character is ready for input. If a character is available from the device, the character must be returned in A. The input routine does not wait for a character to be ready; it simply returns the current status of the device. This permits multitasking to operate without hanging up the system.

The device driver's output routine must accept the characters, in register A, to be sent to the device. When called, this routine should test the status of the device and send the characters to the device when it is ready. In MS-DOS, device 4 is configured to communicate over COM1:, the first RS-232 device.

Chapter Six: 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 Chapter 7). 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 the compiler.

TICS

Similarly, if multitasking is not being used in a program, don't specify TICS ON. This command is essential for multitasking programs without hardware interrupts and for MS-DOS systems using multitasking. It is useless for non-multitasking programs, or those CP/M multitasking programs which use hardware interrupts, and adds extra code to the object file.

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 8 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. 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.

PART TWO

User Reference

Chapter Seven: Direct Commands

Direct commands are those commands given to MTBASIC to control its operation. Direct commands are not part of MTBASIC programs.

BYE

BYE exits MTBASIC and returns to the operating system.

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. If the program requires tics (software interrupts) to run, be sure to specify TICS ON before doing COMPILE. Similarly, specify NOERR or ERROR, if needed, before COMPILEing.

CONSOLE

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

DISK COMPILE

DISK COMPILE produces a stand alone, executable file of the program currently in memory. It generates a .COM file. The object code is generated to disk, and is <u>not</u> left in memory, so a GO command will not cause the DISK COMPILEd program to be executed. The only argument to DISK COMPILE is the file name *without* an extension; MTBASIC will automatically supply an extension of .COM.

When the DISK COMPILE is finished, it prints the start and end addresses of the compiled program., and the address of the bottom of the

variable storage.

DISK COMPILE can be used to compile programs which are too large for MTBASIC to run in its normal interactive mode. If the error message "NOT ENOUGH MEMORY TO COMPILE PROGRAM" occurs while using RUN or COMPILE, then use DISK COMPILE.

Like RUN and COMPILE, DISK COMPILE should be commanded after TICS ON or TICS OFF, ERROR or NOERR, etc.

This example demonstrates LOADing a file and DISK COMPILEing it. Note that the program name must not be in quotes.

```
LOAD PROG.BAS
DISK COMPILE PROG
```

On CP/M systems, DISK COMPILE compiles from a Basic file on disk. All Basic code resident in memory will be deleted. The argument to DISK COMPILE must be the name of the source code file without an extension (the file *must* have an extension of .BAS). DISK COMPILE will generate a file of the same name but an extension of .COM. If an error is detected, the line in error will remain in memory so you can examine it via LIST.

```
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 offline 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
```

```
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, TICS ON, TICS OFF, etc., 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.

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 a DISK COMPILEd 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 recompile the program).

LIST **[line 1, line 2]**

LIST displays the program currently in memory on the system's console. If a PRINTER 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
```

LOAD **filename**

LOAD brings the indicated file into the compiler from disk. 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 assumes an extension of .BAS. 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.

Programs can be created offline with a text editor or word processor and then brought into MTBASIC with the LOAD command. Remember to put an END after the program and to use the word processor in non-document mode.

LOAD allows the imbedding of direct commands other than LOAD and SAVE. A complete environment can be edited to a disk file, and, by typing "LOAD filename", the program can be loaded, compiled, and run.

NEW

NEW erases 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> . (Control-C aborts a running program, so a program compiled

under NOERR can't be killed.) 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 MTBASIC or the 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.

PRINTER

The PRINTER command causes all output sent to the console to also go to the CP/M LST: or PC-DOS LPT1: device. This includes output generated by the program (via PRINT and FPRINT statements), as well as output during direct commands (such as LIST).

CONSOLE disables PRINTER.

RUN

RUN compiles and executes a program. The program is not executed if any compilation errors are found.

SAVE filename

SAVE stores the program currently resident in the compiler to disk. 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 and DISK COMPILE). 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.

STATUS

STATUS displays the start and end address of the compiled program,

along with the amount of free memory and the starting address of the variables used in the program. The space between the end of the program and the start of the variable storage may be used for assembly language routines.

TICINT <address> (MS-DOS only)

Some IBM PC compatible computers use a different interrupt vector than the IBM for their tic source. As a result, MTBASIC's multitasking may not operate properly (or at all) on these computers. If this is the case, you can force MTBASIC to use the proper interrupt vector by issuing the TICINT direct command *before* compiling your program.

The argument to TICINT is the real time clock tic interrupt vector for your computer. For instance, on the IBM this vector is 8 (default setting - IBM users don't need to use TICINT). On the Sanyo 550 this vector is $F8. Example:

```
TICINT $F8
```

This example sets up the interrupt vector for a Sanyo computer.

TICINT need be issued only when you enter MTBASIC when multitasking. It does not have to be issued every time you compile, unless you exit to DOS and return to MTBASIC.

TICS <ON or OFF>

Under CP/M, the TICS commands turn the software tic source on or off for multitasking. If a source of hardware interrupts suitable for tics exists, then the TICS ON is not necessary. TICS ON is required whenever a multitasking program is executed without hardware interrupts. Note that the WAIT statement requires a source of tics, even if it is used in a single task program. When MTBASIC starts, software tics are off, so they must be explicitly turned on (TICS ON) if needed.

Under MS-DOS, TICS ON connects the source of hardware clock tics to MTBASIC's multitasking software. TICS ON therefore provides the computer with a true hardware tic source.

TICS ON and TICS OFF affect the compilation of the program, so they must be specified before the program is DISK COMPILEd or RUN.

VARIABLE address (CP/M only)

The VARIABLE direct command should only be used by advanced programmers generating code to be placed in a ROM. VARIABLE is used to specify the location of RAM memory in the target system. Normally, MTBASIC assigns variables and temporaries to memory right below CP/M. VARIABLE is used to alter this assignment.

The argument (**address**) must be the address of the END of the target system's RAM. MTBASIC will assign storage from this address on down. The address must be in the range of $8000 to $FFFE. $FFFF is illegal. MTBASIC makes calls to CP/M for its I/O, so don't assign RAM storage on top of CP/M unless the target system doesn't have CP/M. (In this case, it may be necessary to intercept CP/M systems calls from address 5.) If no address is specified, VARIABLE will assign memory starting from the bottom of CP/M (MTBASIC's normal RAM assignment).

VARIABLE only takes effect with programs compiled via DISK COMPILE. DISK COMPILE prints the variable start address when it is finished, so you can determine how much RAM is needed.

If VARIABLE is issued with an argument, all size checking is disabled, so the "NOT ENOUGH MEMORY TO COMPILE PROGRAM" error will not be issued. Be sure your code doesn't run into your RAM!

Examples:

VARIABLE $BFFF causes variables used by the program to be assigned from address $BFFF and down in memory.

VARIABLE causes variables used by the program to be assigned from the bottom of CP/M and down in memory.

Note that all .COM files produced by DISK COMPILE are ROMable. The program space starts at address 100H and extends to the address printed at the completion of the DISK COMPILE. Everything between these two addresses must be included in the ROM.

Chapter Eight: Components of Statements

A 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).

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=.0123 (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 6). 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 provides a facility for using hexadecimal constants. Hexadecimal constants are specified with a leading dollar sign. $1AB represents the hexadecimal constant 1AB.

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 Nine: Statements

A statement is an instruction in a program which specifies what action the program will take.

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. Each statement must be 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. Obviously, POKEing into the compiler itself will cause unpredictable results. 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                    This is Z-80 code.
20 DATA $21,$05,$00,$C9
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:

Mode 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!

In MS-DOS systems, the CALL statement generates an indirect intersegment call to the data segment. A long return must therefore be used to return from it. Only variables should be used as arguments, since all variable addresses will be in the data segment.

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 thru an intermediate variable, as follows:

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

A routine called via CALL must restore the segment registers to the values they had on entry before returning. In a multitasking program, DON'T change the segment registers without disabling interrupts.

The CALL statement and its argument list are compiled in MTBASIC's code segment. The actual arguments are in its data segment. When your assembly program gains control the stack will have a long return address on it which is the segment:offset of the start of the argument list. The argument addresses are offsets within MTBASIC's data segment. DS: will contain the proper segment of the offsets. Since a long call is used to get to your routine, CS: will probably not have the right segment value for the argument list; extract the segment from the stack.

CANCEL <task>

CANCEL stops the specified task from being started again when the schedule interval given in the RUN statement elapses. CANCEL does *not* abort the task - only EXIT or STOP can stop the execution of a task. When a RUN 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 RUN statement.

CANCEL is useful when a task need only be run once. The task CANCELs itself, as in the example below.

```
10 INTEGER I
20 RUN 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
```

CHAIN

Compiled programs can start other compiled programs using the CHAIN statement. CHAIN loads the current file (which must be a .COM file) under the current file number (see FILE), deletes the current program from memory, and starts the just loaded one. Any .COM file can be chained to. To CHAIN to another MTBASIC program, compile it separately and save it to disk with DISK COMPILE. Then CHAIN to it, as follows (assume it was saved under the name PROG2.COM):

```
10 OPEN 0,1,"PROG2.COM"
20 FILE 1
30 CHAIN
```

Parameters can be passed between CHAINed MTBASIC programs through variables. If both the calling and the called program define variables in the same order, the variables will stay in memory during the CHAIN. For example, side-by-side listings of a calling and a called program are given below. Variables I,J, and K$ and L, M and N$ will be common between the programs, since they are defined in the same order in each program.

```
5  REM PROG1.COM
10 INTEGER I,J
20 STRING K$(30)
30 INPUT I,J
40 K$="MESSAGE TO PROG2.COM"
```

```
5  REM PROG2.COM
10 INTEGER L,M
20 STRING N$(30)
30 PRINT L,M
40 PRINT N$
```

```
50 OPEN 0,1,"PROG2.COM"
60 FILE 1
70 CHAIN
```

Since the order of definition defines the commonality of the variables, they don't need to have the same name in each program.

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, since the last record may not actually be written until the CLOSE is issued. Errors, if any, are returned by the ERR function. Examples:

```
10 OPEN 0,1,"FILE"
20 CLOSE 1
```

```
10 OPEN 1,1,"TEST"
20 FILE 1
30 PRINT "Message to file"
40 CLOSE 1
```

CLS

CLS is the same as ERASE. It erases the contents of the CRT. MTBASIC must be configured before using CLS (chapter 3).

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:

```
MS-DOS                  CP/M
110 CODE $90            110 CODE $21,0,0
120 RETURN              120 CODE $3E,0
                        130 RETURN
```

CP/M only 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.

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.

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 10 for more information.

DEFSEG [<address>]

DEFSEG changes the segment that PEEK, POKE, and CALL operate on. Normally, if no DEFSEG is issued, these three statements all access MTBASIC's normal data segment.

DEFSEG expects one optional argument. If no argument is given, it restores the default segment to MTBASIC's normal data segment. If an argument is given, it must be in segment-register format, i.e., the low 4 bits of the 20 bit effective address are discarded, resulting in a 16 bit segment value.

Example:

```
10 DEFSEG $B000     Set to segment B000 (address B0000), which is
                    the video memory buffer in the IBM PC.
20 DEFSEG           Restore normal default segment
```

After a DEFSEG is issued, all PEEK, POKE, and CALL statements will reference the segment defined in the DEFSEG.

DELETE <file number>

The DELETE statement deletes the file currently opened under <file number>. Errors, if any, are returned by the ERR function. For example:

```
10 OPEN 0,1,"TRASH"
20 DELETE 1
```

will delete the file called "TRASH".

ERASE

ERASE clears the contents of the CRT. 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 RUN 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.

```
20 INTEGER I
30 I=0
40 RUN I,100
50 IF I<> 10 THEN 50
60 CANCEL 1
```

```
10 INTEGER 1
20 I=0
30 PRINT "This is the beginning."
40 RUN 1,1000
50 RUN 2,200
60 RUN 3,300
70 IF I=10 THEN STOP
80 GO TO 40
90 TASK 1
100 FILE -1
110 PRINT "MESSAGE TO PRINTER"
120 EXIT
130 TASK 2
140 FILE 0
150 PRINT "FIRST MESSAGE TO CONSOLE"
160 EXIT
170 TASK 3
180 PRINT "SECOND MESSAGE TO CONSOLE"
190 I=I+1
200 EXIT
```

MS-DOS or PC-DOS ONLY COM1: Port

MTBASIC comes configured to access the IBM PC's first serial port, COM1:, when FILE 4 is issued. Since MS-DOS does not provide a call to determine the status of COM1:,MTBASIC does a BIOS call to access this port. If you're using an IBM compatable, then this routine may not work. In this case, you'll have to patch locations 106H and 109H as described in Chapter 5.

FNEND

The FNEND statement marks the end of a user defined function. Also see DEF and Chapter 10.

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 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_____________  |_____________outer loop
40 PRINT K,J                 |_____|_____inner loop
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 needed. Leading zeros are printed.

In Prints n integer digits. Leading zeros are converted to blanks before being printed.

Sn Prints n characters of a string.

Xn Prints n spaces

Fm.m 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.

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

Examples:

```
10 FPRINT "H3,X1,F3.4",X,Y
```
if X=$2AB and Y=123.123456, then the output will be

```
2AB 123.1234
```

```
20 FPRINT "S5,X1,S4",A$,B$
```
if A$="Hello" and B$="John Doe", then the output will be

```
Hello John
```

```
30 FPRINT A$,B,C
```
if A$="I1,X1,I1", B=3 and C=9, then the output will be 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
```

GO TO <line number>

GO TO transfers control to the line number given as its argument. Example:

```
10 GO TO 20
20 GO TO 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
```

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.

Examples:

This 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. Even without hardware interrupts, task 1 runs during the INPUT.

```
TICSON                                  a direct command
20 INTEGER K
```

```
30 RUN 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) thru A(10) may be referenced.

Example:

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

INTMODE <digit> (CP/M only)

This statement is used for processing hardware interrupts. <Digit> (which must be between 0 and 2) sets the processor's interrupt mode. Interrupt modes are:

0: In this mode, the interrupting device places any instruction on the address bus. The first byte of a multi-byte instruction is read during the interrupt acknowledge cycle. Subsequent bytes are read by a normal memory read sequence.

1: In this mode, the CPU will execute a call to location $0038.

2: This mode allows an indirect call to any location in memory. The upper 8 bits of the address are to be in register I, and the lower 8 bits are to be supplied by the device that interupted the CPU. Note that the VECTOR and JVECTOR statements expect the address to be in page zero (locations $0 to $FF).

The INTMODE must be issued before interrupts are received.

Using INTMODE instructions requires an understanding of hardware interrupts that is beyond the scope of this manual to explain.

INTON and INTOFF

These statements enable and disable software and hardware interrupts. They are useful for running a top priority task without being interrupted. Do not do I/O when interrupts are off (via INTOFF) if your BIOS requires hardware interrupts to service the devices.

The following example program runs both tasks during the first loop (from 1 to 100). During the second loop, interrupts are disabled, so only the lead task executes.

```
10 INTEGER K
20 RUN 1,100
30 FOR K=1 TO 10
40 PRINT K
50 NEXT K
60 INTOFF
70 FOR K=100 TO 110
80 PRINT K
90 NEXT K
100 STOP
```

```
110 TASK 1
120 PRINT "Interrupt!"
130 EXIT
```

JVECTOR <address>, <task> (CP/M only)

JVECTOR, like VECTOR, is used to link a source of hardware interrupts into MTBASIC. JVECTOR puts a jump to the task, whereas VECTOR just stores the address of the task. In other words, JVECTOR is used with mode 0 and mode 1 interrupts, but VECTOR is used with mode 2 interrupts. When JVECTOR is executed, the starting address of the specified task is placed in a JMP instruction at the address specified. Placing a JMP to a specific task at this address will cause the task to start when the interrupt is received. This is useful for device handlers. JVECTOR 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 INP or PEEK functions, or the POKE or OUT statements allow the user to service the device.

If task 0 is specified in the INTMODE statement, this refers to the source of hardware tics for the multitasking.

The following example program responds to an interrupt RST 7 by placing a jump to task 1 at location 38H, which is the RST 7 address.

```
10 JVECTOR $38,1
20 INTMODE 1
30 INTON
40 GO TO 40
50 TASK 1
60 PRINT "Interrupted"
70 EXIT
```

The next program illustrates a method of connecting a source of hardware interrupts to generate MTBASIC's tics. Since task 0 is specified in the JVECTOR statement, a link to the tic logic will be generated. This example assumes that the interrupt is a mode 1 RST 7.

```
10 JVECTOR $38,0
20 INTMODE 1
30 INTON
40 RUN 1,100
50 GO TO 50
60 TASK 1
70 PRINT "Task 1 - started by hardware tics"
80 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.

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

ON ERROR <line number>

When MTBASIC encounters a file error 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 a file error is found. The line number specified in the ON ERROR will receive control when a file 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$
```

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

The OPEN statement is used to assign a file number to a disk 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.

<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. <File number> is a number from 1 to 3 which is then used to reference the file in FILE statements. Up to 3 files may be open at any one time. <File name> is an ASCII string containing the file name.

The following are examples of valid OPEN statements:

```
10 OPEN 0,1,"FILENAME"      OPENS FILENAME for READ operations
20 A$="STUFF"
30 OPEN 1,2,A$              OPENS STUFF for WRITE operations
```

OUT <expr 1>, <expr 2>

The OUT statement sends the byte <expr 2> to output port <expr 1>. No check is made to see if anything is connected to the port. Out sends only the lower 8 bits of <expr 2> to the port.

The following program sends the letters A through Z to output port 1:

```
10 INTEGER K
20 FOR K=$41 TO $5A
30 OUT 1,K
40 NEXT K
```

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 INTEGER K
20 POKE $8700,$C9
40 STOP
```

PRINT <expr>, [<expr>...]

PRINT sends output to the currently active device or file, based on the setting of the last FILE 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.

For example:

PROGRAM	OUTPUT
10 PRINT "Bagels are delicious."	Bagels are delicious.
10 PRINT 1;2	12

PUT

PUT is used in conjunction with FIELD to perform random I/O. FIELD defines the format of a record; PUT writes a record to disk.

PUT takes no arguments. Every time PUT is executed, the next record is written to the disk. The SEEK statement can be used to write to any record in a file.

When PUT is executed, the current value of the variables defined in the FIELD are written to disk. See chapter 5 for more discussion of random I/O.

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.

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.

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) thru A(10) may be referenced.

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.

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. The SEEK statement can be used to read from any specific record in a file. Chapter 5 contains more information on random file I/O.

RUN <task number>, <schedule interval>

RUN, not to be confused with the direct command of the same name, is used within a multitasking program to start execution of a task. The specified task in <task number> starts execution 1 tic after the RUN 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.

For example:

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

Note that no tasks will run unless TICS ON is specified.

SEEK <record number>

SEEK causes the pointer in the current file (the last one given in a FILE statement) to go to the <record number> specified. Errors, if any, are returned by the ERR and ERR$ functions. SEEK allows random access to any part of a file. Records are all 128 bytes long unless redefined via the FIELD statement.

For example, the following program SEEKs five records into the file "TEST" (20 bytes), and then reads the sixth record into A$:

```
40 OPEN 0,1,"TEST"
50 FIELD 4,A$
60 SEEK 5
70 RGET
```

STOP

This statement stops execution of the program. If the compiler is not resident, that is if you are executing a file made with the DISK COMPILE direct command, then the program returns to the operating system.

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. No more than 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.

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

```
10 RUN 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 file (as specified by a FILE statement), whether it is console, disk, or printer.

Example:

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

VECTOR <address>, <task>

VECTOR, like JVECTOR, is used to link a source of hardware interrupts into MTBASIC. VECTOR is used with MS-DOS or Z-80 interrupt mode 2. When VECTOR is executed, the starting address of the specified task is placed at the address specified. Z-80 interrupt mode 2 causes an indirect CALL to an address in page 0 memory. The Z-80 LD I,A is not supported, so only page 0 vectors may be used, unless an LD I,A is executed via an assembly language subroutine.

MS-DOS systems use only a single interrupt mode, which is functionally equivalent to Z-80 mode 2.

Note that MTBASIC does not provide hardware support of an interrupting device. If the device needs resetting, or any other interaction, the INP or PEEK functions, or the POKE or OUT statements allow the user to service the device.

If task 0 is specified, this refers to the source of hardware tics for multitasking (see JVECTOR).

The following program responds to a vectored interrupt to address 60H by placing the address of task 1 at location 60H.

```
10 VECTOR $60,1
30 GO TO 30
40 TASK 1
50 PRINT "Interrupted"
60  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. When a WAIT statement is executed interrupts are re-enabled (i.e., an INTON is simulated), since a WAIT without interrupts will never terminate (WAIT counts tics).

If TICS ON has not been specified, MTBASIC assumes that a source of hardware tics exists. In this case, if all tasks in a program are in a WAIT simultaneously, then <Control-C> will not kill the program until one of the tasks finishes its WAIT.

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
```

Remember to type TICS ON before RUNning this program.

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 re-sized after being framed.

This example creates a window, frames it, and clears it.

```
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.

On MS-DOS versions of MTBASIC, most computers can make use of the special characters in the display set to draw really nice window frames. If the special characters CHR$($C4) and CHR$($B3) are used, MTBASIC will draw continous lines with true corners. For example:

```
10 WFRAME CHR$($C4),CHR$($B3)    MS-DOS only
```

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.

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

Example:

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

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.

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.

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.

Chapter Ten: 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.

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
```

ADR(<variable name>) INTEGER

Returns the address of the variable <varname>. 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

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

Example

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

CHR$ (<expr>) STRING

Returns a one character equivalent to 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
```

CVI(<string>) INTEGER

Converts the binary string argument to integer.
Example
```
10 PRINT CVI("AB")
```

CVS(<string>) INTEGER

Converts the binary string to real
Example
```
10 PRINT CVS("ABCD")
```

ERR INTEGER

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

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 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) ' (=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 file. INPUTand 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 FILE I
40 FOR I= 1 TO 128
```

```
50 T(I)=GET
60 NEXT I
70 CLOSE 1
```

INP (<expr>) INTEGER

Reads input port <expr>. If the specified port number is greater than 255, the lower 8 bits of the port number are used. An 8 bit value is returned.

Example

```
10 INTEGER P
20 PRINT "Which port (0-255)?"
30 INPUT P
40 IF P>255 THEN 20
50 PRINT "I/O port # ";P;" holds a ";INP(P)
60 GO TO 20
```

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 file number is active. 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). KEY returns an integer representation of the character (i.e., its ASCII value). If a string is needed, convert it with CHR$.

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

Instead, set a variable to KEY and test and print the variable, as in the example. In MS-DOS systems, the special function keys return 2byte strings. If read via KEY, the first byte will be returned as $FF, followed by the normal 2nd byte code.

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."
```

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$
```

MKI$(<integer>) STRING

Converts an integer to a binary string

Example
```
10 STRING A$
20 A$=MKI$(123)
```

MKS$(<real>) STRING

Converts a real number to a binary string.

Example
```
10 STRING A$
20 A$=MKS$(1.23)
```

PEEK(<expr>) INTEGER

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

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

RND INTEGER

Generates a pseudorandom 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
```

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 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
```

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$)
```

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 no 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 Eleven: 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 nonzero 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.

Appendix A: Direct Commands

BYE	Exits MTBASIC and returns to CP/M
COMPILE	Compiles a program without running it
CONSOLE	Directs output to console (disables PRINTER)
DISK COMPILE	Compiles a program to a .COM file
END	Marks the end of a source program file
ERROR	Turns on runtime error checking (default is ON)
GO	Starts an already compiled program running
LIST	Displays the program code
LOAD	Reads a source file from disk
NEW	Erases the current program
NOERR	Turns off runtime error checking
PRINTER	Sends output to the printer (disables CONSOLE)
RUN	Compiles and runs a program
SAVE	Saves a program's source code to disk
STATUS	Displays the amount of available memory
TICINT	Sets tic interrupt vector (MS-DOS only)
TICS	Turns software interrupts on or off (default is OFF)
VARIABLE	Sets RAM addresses (CP/M only)

Appendix B: Summary of Statements

CALL	Starts an assembly language subroutine
CANCEL	Stops a task
CHAIN	Load and execute another COM file
CLOSE	Closes a file
CLS	Erase the CRT
CURSOR	Positions the cursor in a window
DATA	Defines a group of constants
DELETE	Deletes a file
ERASE	Clears the entire CRT
EXIT	Terminates a task
FIELD	Specify a random record format
FILE	Selects an I/O device
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
INTMODE	Defines interrupt mode (CP/M only)
INTON	Turns interrupts on
INTOFF	Turns interrupts off

JVECTOR	Defines interrupt vector (CP/M only)
NEXT	End of a FOR/NEXT loop
OFF ERROR	Disable error trapping
ON ERROR	Start error trapping
OPEN	Opens a file
OUT	Output to an I/O port
POKE	Modifies a memory location
PRINT	Outputs data
PUT	Send a record to a random file
RANDOMIZE	Seeds the random number generator
READ	Gets data from a DATA statement
REAL	Defines floating point variables
REM	Comment, also denoted by ! and
RETURN	Return from a subroutine
RGET	Get a record from a random file
RUN	Starts a task going
SEEK	Random file I/O record position
STOP	Halt a program
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	Links to interrupt vector
WAIT	Delays a task's execution
WCLEAR	Erases a window

WFRAME	Draws an outline around a window
WINDOW	Defines a window
WSAVE	Saves the contents of a window
WSELECT	Selects a window
WUPDATE	Restores a saved window

:	Separates multiple statements on a line

Appendix C: Summary of Functions

ACOS	Arccosine
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
CVI	Convert a binary string to integer
CVS	Convert a binary string to real
ERR	Returns error numbers
ERR$	Returns error messages
EXP	Compute e**x
GET	Returns one character from the current file
INP	Reads input expression from input port
KEY	Returns one ASCII value from console
LEN	Returns the length of a string
LOG	Natural log (base e)
MID$	Returns part of a string

MKI$	Convert an integer to a binary string
MKS$	Convert a real to a binary string
PEEK	Returns the contents of a memory address
RND	Generates random numbers
SIN	Sine
SQR	Square root
STR$	Converts numbers to strings (converse of VAL)
TAN	Tangent
VAL	Converts strings to numbers (converse of STR$)

Appendix D: 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. Check to be sure you specified the correct disk drive.

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 FILE NUMBER: occurs if a file number is specified which is not -1 to 6.

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.

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 disk 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 in the following circumstances:

A GO command is entered, but no RUN or COMPILE has taken place.

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. DISK COMPILE can be used in case there is not enough room in memory for both the source and object code.

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:
receiving a hardware interrupt which was vectored
to a non-existent task.
trying to RUN a non-existent task.
trying to RUN TASK 0.
trying to use more than 9 TASK statements.

TOO MANY VARIABLES: occurs if more than 255 varables are used. If you need more than 255 variables, break the program up into several modules and then CHAIN them together, or use 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.