CHAPTER

7

PROGRAMMING

What is a program?

The traditional description of a program is a task that you want the computer (the *Motion Coordinator*) to perform. The task is described using statements written in the Trio BASIC language which the *Motion Coordinator* can understand.

A program is simply a list of instructions to the *Motion Coordinator*, some of these instructions have a dedicated function to be performed by the controller, others control the program flow, the sequence in which instructions are actually executed.

Statements in your program must be written using a set of rules known as 'Syntax'. You must follow these rules if you are to write Trio BASIC programs. Trio BASIC instructions are divided into the following types:

- Instructions
  - Program Flow
  - Controller Specific
- Identifiers
  - Labels
  - Data Storage

Controlling the Sequence of Events

In order to write a program we must break the function of our system down into logical operations which the controller must perform. As we are not able to solve every problem in a purely linear manner, we need more control of the ‘flow’ of the program instructions, for example to make a decision and decide whether or not certain instructions need to be executed, or to perform a certain task several times. In programming terms we refer to these concepts as **SEQUENCE**, **SELECTION** and **ITERATION**.

Sequence

The ability to process a series of instructions, in a logical order, and to control the flow by branching to another part of the program.

Normally, a program executes statements in sequence starting at the top. In order to branch between different sections of the program we need to be able to identify specific sections of the code. Labels are used as place markers to indicate the start of a routine, or the target for the 'branch' instructions, **GOTO** and **GOSUB**.

It is useful to split your program up into a series of routines, each of which handles a particular funtion of the machine. The **GOSUB** command will jump to a label and continue from its new location. When the program encounters a **RETURN** command, the program will jump back to the **GOSUB** from where it originally came. Take the following example:

```
PRINT "Hello"
GOSUB a_subroutine
STOP

a_subroutine:
  PRINT "World"
RETURN
```

The program will print the "Hello" text to the terminal window, then jump to the line of the program labelled **'a_subroutine'** and continue execution. The next command it finds will print "World". The **RETURN** command then returns the program to the point it left, where it then proceeds onto the next command after the **GOSUB** command which in this case is the **STOP** command, which halts the execution of the program.

The **GOTO** command does not remember where it jumped from and will continue running from its new location permanently. This might be used for example, if we have a certain process which needs to be performed when shutting down a machine, we might jump directly to that routine:

i.e. **GOTO shut_down**

Trio BASIC instructions:

Labels, GOTO, GOSUB, RETURN, STOP

Selection

Commands that enable us to selectively execute instructions depending on certain criteria being met.

Example: **IF** we have made a complete batch **THEN** stop the machine

Trio BASIC Instructions:

```
IF … THEN … ELSEIF … ENDIF
ON ... GOTO
ON ... GOSUB
```

Iteration

To repeatedly execute one or more commands automatically, either for a specified number of times, or until a certain condition is met or event occurs.

Example

```
REPEAT
  GOSUB index_conveyor
UNTIL IN(product_sensor)=ON
```

Trio BASIC instructions:

```
FOR … TO … STEP … NEXT
REPEAT … UNTIL
WHILE … WEND
```

FOR..NEXT Statements

The **FOR .. NEXT** commands are used to create a finite loop in which a variable is incremented or decremented from a value to a value.

Example:

```
FOR t=1 TO 5
  PRINT t;" ";
NEXT t
PRINT "Done"
```

The output to the screen would read:

```
1.0000 2.0000 3.0000 4.0000 5.0000
```

The program would set the variable **t** to a value of 1 and then go to the next line to **PRINT**. After the print, the **NEXT** command would return the program to the **FOR** command and increment the value of T to make it 2. When the **PRINT** command is used again, the value of T has changed and a new value is printed. This continues until T has gone from 1 through to 5, then the loop ends and the program is permitted to continue. The next command after the **NEXT** statement prints "Done" to the screen slowing the program has left the loop.

You can also use for-next loops to create a loop within a loop, as the following example shows:

```
FOR a=1 TO 5
  PRINT "MAIN A=";a
  FOR b=1 TO 10
    PRINT "LITTLE B=";b
  NEXT b
NEXT a
```

The **FOR..NEXT** statement loops the main A variable from 1 to 5, but for every loop of A the **FOR..NEXT** statement inside the first loop must also loop its variable B from 1 to 10. This is known as a nested loop as the loop in the middle is nested inside an outer loop..

Such loops are especially useful for working on array data by using the variables that increment as position indexes for the arrays. As an example, we could perform a sequence of absolute moves like this:

```
FOR y=12 TO 1 STEP-1
  FOR x=10 to 120 STEP10
    MOVEABS(x,y)
  NEXT x
NEXT y
```

As can be seen,the for-next loop can count down as well as step in value, insted of simply incrementing the loop counter.

Controller Functions

The specific commands, which instruct the processor to perform a predefined function or operation. Each instruction will be assigned its own *reserved word* in the language.

For example the **PRINT** instruction in Trio BASIC is used to display a message or numeric value on the computer screen or another output device, such as a printer.

Instructions vary in complexity and will take a variety of formats. Some will be a single keyword with a clearly defined function, such as **CANCEL** or **STOP**, whereas others may take one or more *parameters* which affect the operation of the command.

examples:

WA(1000)	wait for a specified time (in milliseconds)
PRINT "Hello"	Display the word "hello" on the terminal screen
GOTO show	redirect the program to the part labelled *show*

Identifiers

Identifiers are the names which the programmer uses to identify (!) things in the program. There are essentially two main types of user-defined identifier, Labels and Variables.

Labels

Labels are used to provide a place-marker in a program. Not only does this make the code more readable, it also enables us to direct the flow of our program to a specific place.

In Trio BASIC, labels are defined by placing a name at the start of the line, followed by a colon (:).

e.g.

```
start:
enter_password:
error_handler:
```

Variables

Variables are storage locations for numeric values. they are called variables as they can be changed at any time. Just like labels, variables can often be given a user-defined name. Anywhere a number is required a variable can be used. Only the first charactors of each variable name are used to identify the unique variable. For example; *Micromanipulator1* is the same as *Micromanipulator2*

Note: Only up to 16 charactors may be used for variable names.

Example: **`batch_size=10`**
would assign a value of 10 to a variable called “batch_size”. Then anywhere in the program that needs to know the value stored can read this value by name.

Trio BASIC *has three different variable types:*

`named variables` These are LOCAL variables - i.e. they are only valid within the task they are defined.

Each process can define up to 1024 named variables .

Example:

```
a=123
SPEED=user_speed
PRINT #3,”Length = “;prod_length[2]
```

`VR() variables` The controller has a global array of 1024 **`VR()`** variables which are shared between tasks (1024 on MC206).

Example:

```
VR(2)=123.4567
```

`TABLE memory` The **`TABLE`** memory is a large array of up to 256k entries depending on the controller type. Normally used to store profiles for the **`CAM`**/**`CAMBOX`** commands.

If the controller features a battery backed memory, **`VR()`** variables and **`TABLE`** memory will be retained when the power is off. For controllers without a battery, e.g., the MC302X, the **`FLASHVR()`** command is provided to store the values in flash eprom memory.

Expressions

An expression is defined as any calculation or logical function which has to be evaluated.   An expression may be used anywhere a number is required, or a logical (TRUE/FALSE) decision. In the case of logical expressions, TRUE is deemed to be any non-zero result.

In programming, the component parts of an expression are known as operands and operators. The *operands* are the values, either specific numbers, or variables. The *operators* are those functions or actions which act on the operands.

Example 1: You can assign the result of an expression to a variable,

`num_widgets = total_length / widget_length`
has three operands, **`num_widgets, total_length`** and **`widget_length`**
and two operators, = (assignment) & / (divide).

Reading the above as simple English would equate to:

Divide the variable **`total_ length`** by **`widget_length`** and assign the result to the variable **`num_widgets`**

Example 2: you could use an expression directly:

`MOVE(widget_length+10) '`(MOVE is a Trio BASIC instruction)

Example 3: Sometimes an expression is used to make a decision..

```
IF batch_count = batch_size THEN GOTO batch_done
```

Parameters

Parameters are special purpose variables, used by the system for configuration and feedback.

Axis Parameters

Each of the axes has its own set of axis parameters which are used to achieve many of the *Motion Coordinator* features. The axis parameters may be floating point or 32 bit integer. The parameters are all set to default values on every power up. Parameters are read from and written to like variables. The Trio BASIC assumes the current BASE axis is the required axis unless the AXIS modifier is used:

```
>>P_GAIN=2
>>P_GAIN AXIS(8)=0.25
>>? VP_SPEED AXIS(2)
```

A list of all the axis parameters is given in chapter 8

System Parameters

Trio BASIC holds a list of parameters which are common for the whole controller. These parameters can be read from and written to like variables. The system parameters are described in chapter 8. Note that as there is only one system there is no modifier for system parameters.

Process Parameters

Trio BASIC also holds a small number of parameters which are held separately for each PROCESS.

These are:

1) **TICKS**
2) **PROCNUMBER**
3) **PMOVE**
4) **ERROR_LINE**
5) **INDEVICE / OUTDEVICE**
6) **BASE**

The process assumed is the current process the command is using, however it is possible to force the controller to read parameters from a specific process with the PROC() modifier.

Example:

```
WAIT UNTIL PMOVE PROC(14)=0
```

Forcing priority of program execution

When a user program is running, it is known as a 'task', or a 'process'. The number of simultaneous processes available is dependant on the controller type. When a program is started, the *Motion Coordinator* will allocate it to a process automatically to make the system easier to use. This will normally be sufficient for most applications, especially when there are less than 4 programs in use.

Allocation of Time

For more complex applications it can be useful to allocate execution priorities to programs. In order to do this we need to understand how the *Motion Coordinator* normally allocates the available processing time:

The default servo period is 1mS. This period is internally divided into 3 time slots of 1/3mS each, which are used internally for processing the servo functions, communications and general 'housekeeping' tasks respectively. The remaining time in each of these slots is used for running the user's application programs.

1ms			1ms			1ms		

Table 1:

Process Numbers

The processes available for programs are identified by numbers, from 1 to the maximum available on the controller. For example, an MC224 can run 14 simultaneous programs. Process 0 is also allocated automatically to the *Motion Coordinator*'s command line interface / *Motion* Perfect connection.

Note: The maximum number of processes available is dependant on the controller type, as shown in the table below.

Controller	Max # Processes	High Priority Processes
MC302X	3	3
Euro205x	7	7,6
MC206x	7	7,6
MC224	14	14,13

The two highest numbered processes (14 and 13 in our example MC224) are allocated a fixed time slot. These are referred to as the "fast" tasks. They should be used for processes which require:

- Guaranteed processing every servo cycle
- A large number of calculations or processing
- Program execution which does not vary in speed as tasks are started or stopped.

Any other processes (including the command line) share the third time slot. Execution speed will therefore reduce as the number of programs running increases. In practice however, a useful execution speed is still obtained. Processes 1..12 are referred to as “standard” tasks.

Programs can be forced to run on a specific process using the commands **RUN** or **RUNTYPE**:

>>RUN "progname",7	Run the named program immediately on specified task
>>RUNTYPE “progname”,ON,7	Assigns start-up mode for specified program

If equal time is required to be given to all programs, the high priority processes (14 and 13) should NOT be used. The time available will then be divided evenly between the remaining processes. The command line and processes 1, 2 & 3 share the remaining third. These programs and the command line use the available time slot with equal priority

Examples: No fast tasks, two standard tasks

1ms			1ms			1ms		
1	2	C/L	1	2	C/L	1	2	C/L

Table 2:

No fast tasks, three standard tasks

1ms			1ms			1ms		
1	2	3	C/L	1	2	3	C/L	1

Table 3:

Two fast tasks, two standard tasks

1ms			1ms			1ms		
14	13	1	14	13	2	14	13	C/L

Table 4:

One fast task, two standard tasks

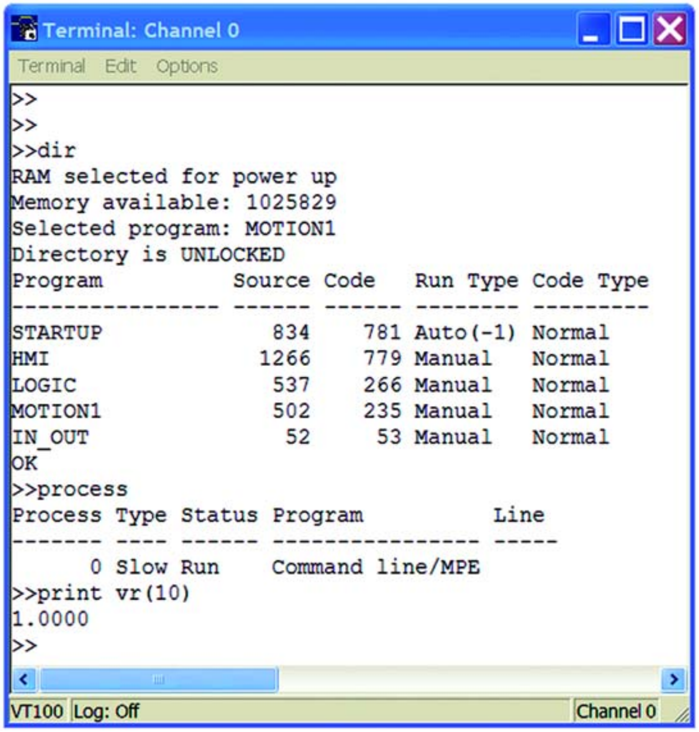
This example shows the case where there is one fast task only. This is the exception to the rule as it is allocated BOTH 'fast' time slots.

1ms			1ms			1ms		
14		1	14		2	14		C/L

Table 5:

Command Line Interface

A "Command Line" interface to the controller can be set up by opening a "Terminal" window in *Motion* Perfect*. The command line interface always uses channel 0.*



```
>>
>>
>>dir
RAM selected for power up
Memory available: 1025829
Selected program: MOTION1
Directory is UNLOCKED
Program          Source Code   Run Type Code Type
---------------- ------ ------ -------- ---------
STARTUP             834    781 Auto(-1) Normal
HMI                1266    779 Manual   Normal
LOGIC               537    266 Manual   Normal
MOTION1             502    235 Manual   Normal
IN_OUT               52     53 Manual   Normal
OK
>>process
Process Type Status Program          Line
------- ---- ------ ---------------- -----
      0 Slow Run    Command line/MPE
>>print vr(10)
1.0000
>>
```

Typing Commands for Immediate Execution

When the controller is waiting for a Trio BASIC command to be typed in it prints the prompt >>

Example: **>>PRINT "HELLO"**

Note: *A line must always be terminated by pressing the ENTER key (<CR>)*

Limitations of the command line

The command line interface is intended to execute single commands. It is not possible to process multiple-statement lines or those commands which control the sequence or 'flow' of a program.

For example, the following type of commands are not available on the command line:

- Loop Instructions:
  FOR..NEXT, WHILE..WEND, REPEAT..UNTIL
- Wait Instructions:
  WA(time), WAIT UNTIL, WAIT IDLE
- Named variables:
  These are local to a program

Attempting to use any of these commands on the command line may produce unpredictable results!

Tip! *The command line features a buffer of the last 10 commands used. This can save a lot of typing on the PC. Pressing the up arrow or down arrow cycles through the buffer.*
If you find a command you do not recognise it was probably put there by Motion Perfect!

Setting Programs to run on power up

Programs can be set to run automatically on power-up using the "Set power up mode..." facility under the "Program" menu. This sets the **RUNTYPE** automatically

Example Typically only ONE program is set to run on power up. This program can then start the others under program control:

```
...body of program
RUN "Prog2"
RUN "Prog3"
...body of program
```

After setting one or more programs to run on power up the project should be set to "Fixed". The programs will then be stored in flash Eprom.

Example Programs

Example 1

```
start:
  TICKS=0
  PRINT "Press a key"
  WAIT UNTIL KEY
  GET k
  PRINT "You took ";-TICKS/1000;" seconds"
GOTO start
```

Example 2

```
'Set speed then move forward then back:
  PRINT "EXAMPLE PROGRAM 2"
  SPEED=100
  ACCEL=1000
  DECEL=1000
  MOVE(250)
  MOVEABS(0)
  STOP
```

Note that the last line stops the program, not the motion. The first line is a comment. It has no effect on the program execution.

Example 3

```
‘Display 16 INPUTS as a row of 1’s and 0’s
REPEAT
  FOR i=0 TO 15
    IF IN(i)=ON THEN
      PRINT "1";
    ELSE
      PRINT "0";
    ENDIF
  NEXT i
  PRINT CHR (13);
  ‘Character 13 will do <CR> without linefeed
UNTIL 0
```