MULTI-BASIC

MULTI-TASKING BASIC

FOR

THE B.B.C. MICROCOMPUTER

BMS SOFT

MULTI-BASIC for the BBC Microcomputer

User Guide

Acknowledgements

The Authors would like to thank the following, without whom neither this document nor the associated program would have been possible.

Bray, Dickens & Holmes for The Advanced User Guide.
Colin Pharo for The Advanced Basic Rom Guide.
Very special thanks to Mark Plumbley for the really excellent Basic Rom Guide.
Extra-special thanks to Phil Taylor, Alf Haills and John Trew, and the staff of Cambridge Microprocessor Systems, for their help in testing, program examples, and their boundless enthusiasm.
Kindest regards also to Denis and Ros, of Westgate Stationary (Newcastle), for their invaluable help and advice in the formatting and printing of this document.

Where in this document the term BBC Microcomputer is used, the Authors are referring to the British Broadcasting Corporation Microcomputer.

B.P.Worrall
J.W.Brown April 1985

Introduction

MULTI-BASIC is an extension to BBC BASIC, providing multi-tasking and control facilities that were hitherto impossible in a 'normal' interpretive BASIC environment. The program embodies very powerful, and perhaps more importantly, user-friendly procedures and functions for the control of multi-tasking and input/output devices on the BBC microcomputer. 'Tasks' may be included as background activities, to be executed concurrently with the main program, giving a secure and easily maintained environment. A task may be activated on an interval and/or event basis i.e. at specified time intervals and/or on a specified event, such as input on a VIA line.

This version for the BBC Microcomputer, is essentially a sub-set of the powerful control language developed for Cambridge Microprocessor Systems (CMS) Ltd., which has over 50 new BASIC keywords for control and access of all the CMS system Eurocards, some of which are documented at the rear of this manual.

Contents

(i) Fitting the ROM and getting started
(iii) How to use this book

Sections

1. Multi-tasking BASIC
2. Keyword descriptions
3. VIA Control
4. System Timers
5. Command Summary
6. Errors & Exceptions
7. DOs, DON'Ts & AVOIDs
8. Further examples
9. System use

Addenda The CMS Euro-card range

ROM Installation.

The program is supplied in ROM form for the BBC microcomputer. The ROM should be inserted into a spare language socket on the BBC machine as per figures 1 or 2. If you are unsure of your ability to do this, then consult your local dealer. Note that MULTI-BASIC is an extension to BASIC, and the BBC BASIC ROM is still required, so don't remove BASIC in the process of fitting MULTI-BASIC !

Getting started.

If not in the default socket, MULTI-BASIC can be initialised with the command:

*MULTI<CR>

or more briefly:

*MU.<CR>

MULTI-BASIC will not function while a Tube connection is current. An attempt at initialisation will cause MULTI-BASIC to display:

Please disconnect your tube

and will not then allow you to proceed. If you are using, e.g. a 6502 2nd processor, then you will need to disconnect the Tube, or power off the 2nd processor.

(Rom fitting diagrams)

Fig. 1

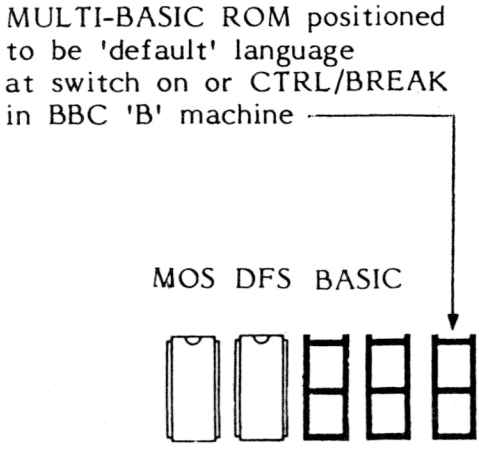
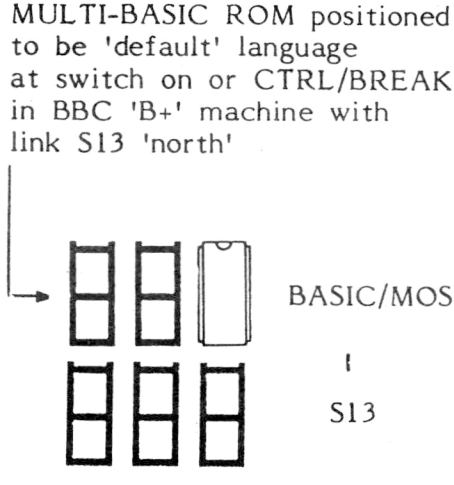


Fig. 2



How to use this Book

The text is presented in sections, with section 2 intended as a 'quick reference' to all keywords in the program. As the BBC user will be primarily interested in Multi-Tasking, it is suggested that section 1 is read first, and that you try some of the simple example programs so as to familiarise yourself with the new keywords and their actions.

The examples we have given throughout this document are intended as guideline 'skeletons' only, and should not be regarded as definitive. Where a section is long enough to justify this, an index to that section is given on the first page(s).

Briefly, each section comprises the following:

Section 1. Multi-tasking

Rationale and use of the concurrency facilities.

Section 2. Keywords

Section 2 is a reference section in which each keyword, its purpose, required parameters and example of use are given, followed by associated keywords. This section is arranged in alphabetic order, and the Authors have attempted to include in each keyword description all relevant information for correct use of that keyword. Because of this approach, there has inevitably been duplication of material across the section, and we beg the indulgence of those users who have the enviable ability to remember from function to function just what is allowable and what is not.

Section 3. Versatile Interface Adaptors

Using/abusing the system VIAs for input/output and Timing.

Section 4. The system Timers

Using BASIC functions to Read/Write the system Timers.

Section 5. Command Summary

A summary of all commands together with allowed parameter types and value ranges allowed. The use of the indirection vectors ! and ? etc.

Section 6. Errors & Exceptions

Error messages generated by MULTI-BASIC, with suggestions as to their cause and cure.

Section 7. DOs and DON'Ts

What you should have done, or not done (when it dosn't work !).

Section 8. Further example programs.

Section 9. System use

Locations and Devices used by MULTI-BASIC.

Section 1. Multi-Tasking

Index to this section.

1. Statement execution time.
2. BASIC statements.
3. System timed-interrupts.
4. BASIC 'tasks'.
5. Some keywords and their actions.
   TASK
   ENABLE
   EVERY
   WHENEVER
7. Example program.
9. Program format & operation.
10. DISABLE
11. OMIT
13. Controls on declared and enabled tasks.
   DECLD
   ENABD
15. BBC Events.
   Use of WHENEVER
17. Example using joystick fire buttons
19. Example of printer spooler
23. Concurrent INPUT.
   Getting text in Real-Time
27. Use of EVENTV.
28. *EVENTSON & *EVENTSOFF.
31. Tackling the unexpected.
   Task overrun
   Overrun detection
33. Cause of overrun.
34. Unserviced events.
35. Escape & Break.

Statement execution time.

BBC BASIC, as we all know, is an interpretive language, i.e. source statements are parsed by the statement parser, and then evaluated into a series of calls to machine-code which perform the actual commands, functions and procedures. All of this takes time of course, and generally speaking, BASIC proves far too slow for most real-time applications which require a very fast response (in the micro-second range). Fortunately, the majority of applications do not need such speed, but work quite happily with response times in the centi-second range. Even at this more tolerant level however, most BASIC interpreters cannot cope - except that is for BBC BASIC. It is a tribute to Acorns' software engineers that they have produced an interpreter so powerful, terse, and with an excellent response time, that simple statements can be executed in a matter of a few centi-seconds. It is a fact that without such speed, it would not be possible, or indeed sensible, to attempt driving the interpreter in a time-shared or concurrent environment. Having absorbed the above, if you still wish micro-second reponse to your programs, then BASIC, even BBC BASIC, is not what you should be using. You will need a compiler-interpreter such as FORTH, or the more laborious assembly language, before such speed is attained.

BASIC Statements

A statement in BASIC has one important peculiarity - it must be terminated !
For instance:

```
10 I% = I%+1
```

is a BASIC statement which is terminated with the end-of-line (carriage-return) character with ASCII code 13, although this does not appear on your listing.

```
20 F% = (I%*2) : G% = 1
```

The above line contains 2 statements, the first terminated with a colon (:), and the second with the end-of-line character.

In MULTI-BASIC, a BASIC statement is considered to be indivisible - that is we cannot break away in the middle of one to do something else. Consider the implications of leaving:

```
G% = 1
```

before completion, to execute another BASIC statement, particularly if the 'break in' statement also made an assignment to G%.

What we can do however, is interrupt the assignment, perform a simple and fairly short machine-code operation, and then return to finish off the statement. As long as we do not affect correct execution of the statement, all will be well. Such interrupts occur frequently on the BBC Microcomputer.

System Timed Interrupts

After the command:

*EVENTSON

(or after initialising with *MULTI)

the User Versatile Interface Adaptor (VIA) generates interrupts every 1 centi-second. During such an interrupt, foreground activity is halted and the interrupt service routine performs certain operations before returning control back to the interrupted program. You need not concern youself on the mechanism - simply on the results of such intrusion. The operations performed by the interrupt service routine are:

a) Increment the system timers 1-5, setting an appropriate flag when a timer 'times out'.

b) Increment any 'enabled' TASK counter and compare with the user-supplied interval - flagging time-out if applicable.

c) Test the user-supplied status addresses and flag where applicable.

d) Enable the transfer of control from the 'foreground' to any or all 'background' tasks depending on the above tests.

The above-mentioned transfer of control will NOT take place until the currently executing foreground statement is finished - remember we remarked earlier that each BASIC statement will be considered indivisible.

BASIC 'tasks'

You will note that we have referred to the word 'task' already. This concept is used so as to clarify the difference between what would normally be an ordinary BASIC program, and those parts of the program we want executed at particular time intervals or when a certain 'event' takes place.

We will refer to the 'normal' part of the program as the FOREGROUND 'task' and all statements within a:

TASK .... EXIT

construct as a BACKGROUND 'task'.
The group of statements making up such a background task may be given a meaningful name.

As with each foreground statement, so each background task is indivisible. In no way then, can one task 'break in' on another. A MULTI-BASIC program will therefore consist of both a foreground task and several background tasks. The latter will be executed whenever an associated time interval has been achieved, or when an associated 'event' has occurred, the background tasks will be slotted between the indivisible foreground statements. The program MUST contain a foreground task, even if this is just a dummy. For instance:

```
500 REPEAT
510 UNTIL 0
```

is a perfectly legitimate foreground task. The program will NOT function when BASIC is in direct mode (not executing a program).

Some keywords and their actions

TASK

A background task is defined as a statement or series of statements within the construct:

```
TASK
:
:
EXIT
```

For example:

```
100 TASK fred
110      I% = I%+1:PRINT I%;
120 EXIT
```

would define the task 'fred'. All 'fred' will do when executed, is increment I% and output its value to the screen.

ENABLE

If you 'RUN' the above program, nothing at all would happen - except that 'fred' would be added to a list of 'current' tasks. To make 'fred' work, we have to tell the system either how often to execute him, or on what system condition (event), and for this the:

```
ENABLE .... EVERY
```

or:

```
ENABLE .... WHENEVER
```

constructs are used.

If we add the lines:

```
130 ENABLE fred EVERY 100cs
```

plus a 'dummy' main program as a foreground task:

```
140 REPEAT
150 UNTIL 0
```

We would find that I% is incremented every 1 second.
Alternatively you may wish 'fred' to be executed only when a certain condition occurs in the system. We term such a condition an 'event', and this could be for example, the signalling of a successful data transfer on a VIA.

Consider the following:

```
200 evnt% = &FE6D0200
210 ENABLE fred WHENEVER evnt%
```

The integer variable evnt% comprises 3 distinct parts when used with WHENEVER.

<address><mask><invert>

where <address> is the 2 most significant bytes of the variable, <mask> is the 8-bit-pattern used in the test, and <invert> an 8-bit-pattern of those bits you wish inverted before the test is made.
If this is as clear as mud, let us explain using the example given.

We know that the User VIA has a base-address of &FE60. The Interrupt Flag Register (IFR), is therefore located at &FE6D, and we wish to test bit 1 of this register. So:

```
<address> = &FE6D   IFR register
<mask>    = &02     Test Bit 1
<invert>  = &00     No inversion required
```

concatenating the above gives &FE6D0200 - quite simple really !

Now the above condition is checked at each timed interrupt (1 centi-second), and if the condition is true (bit 1 set), 'fred' would be executed as a background task at the earliest opportunity.

We present here a simple program containing 8 background tasks, to illustrate the points already made.

```
 10 MODE 7
 20 REM Switch Cursor Off
 30 VDU 23;8202;0;0;0
 80 REM ******************************
 90 REM         Background Tasks
100 TASK joe
110    B%=B%+1
120    PRINT TAB(15,07);" joe  says B% = ";B%;
130 EXIT
140 TASK john
150    C%=C%+1
160    PRINT TAB(15,09);" john says C% = ";C%;
170 EXIT
180 TASK barry
190    D%=D%+1
200    PRINT TAB(15,11);" barry says D% = ";D%;
210 EXIT
220 TASK alan
230    A%=A%+1
240    PRINT TAB(15,13);" alan says A% = ";A%;
250 EXIT
260 TASK fred
270    I%=I%+1
280    PRINT TAB(15,15);" fred says I% = ";I%;
290 EXIT
300 TASK bill
310    J%=J%+1
320    PRINT TAB(15,17);" bill says J% = ";J%;
330 EXIT
```

```
340 TASK jim
350    K%=K%+1
360    PRINT TAB(15,19);"jim  says K% = ";K%;
370 EXIT
380 TASK mick
390    L%=L%+1
400    PRINT TAB(15,21);" mick says L% = ";L%;
410 EXIT
420 REM ****************************
430 REM     Define some variables
440 B%=0
450 C%=0
460 D%=0
470 A%=0
480 I%=0
490 J%=0
500 K%=0
510 L%=0
520 F% = 0
530 lentime% = 0
540 start%=0
550 fini%=0
560 REM ****************************
570 REM     Set up the screen
580 CLS
590 PRINT"            ********************"
600 PRINT"            * Foreground Task *"
610 PRINT"            *                  *"
620 PRINT"            ********************"
630 PRINT" ****************************************";
640 PRINT" *                                      *";
650 PRINT" *          Background Tasks            *";
660 FOR y%= 1 TO 16
670 PRINT" *                                      *";
680 NEXTy%
690 PRINT" ****************************************";
700  REM ****************************
710 REM Enable the Tasks
720 ENABLE joe EVERY 3200cs
730 ENABLE john EVERY 1600cs
```

```
740 ENABLE barry EVERY 800cs
750 ENABLE alan EVERY 400cs
760 ENABLE fred EVERY 200cs
770 ENABLE bill EVERY 100cs
780 ENABLE jim  EVERY 50cs
790 ENABLE mick EVERY 25cs
800 REM ******************************
810 REM          Foreground Task
820 REPEAT
830 F% = F%+1
840 PRINTTAB(13,2);"Time=";TIME;
850 UNTIL 0
860 REM ******************************
```

Referring to the block of statements on lines 720-790, you will see that we have 'enabled' the 8 tasks so that:

```
mick  is executed every quarter of a second
jim    "      "        "   half    "  "    "
bill   "      "        "   second
fred   "      "        "    2  seconds
alan   "      "        "    4     "
barry  "      "        "    8     "
john   "      "        "   16     "
joe    "      "        "   32     "
```

and the important points to note are:

a) When two or more tasks time out together, each timed-out task is executed.

b) The tasks are executed in the order in which they are declared. For instance, after 32 seconds have elapsed, ALL the tasks will be executed starting with 'joe', and finishing with 'mick'. You should bear this in mind when attempting to pass parameters between tasks (by updating an integer variable for instance).

c) The overall program time 'stolen' from the foreground will vary depending on how many tasks have timed out.

The program is arranged in the following order:

1) Selecting the screen mode and switching off the cursor

2) Declaring the background tasks.

3) Initialising variables and setting up the screen.

4) Enabling the tasks.

5) Executing the foreground task.

Generally speaking, this is the order in which you should present your code in a program. In the above example we did NOT want tasks which write to the screen to execute while we were still setting it up.

DISABLE

Having got all the tasks executing nicely, how do we stop one ? We may, for example, wish to execute a particular task after a certain time interval only once - or enable it at only specific periods. For this purpose we use the command DISABLE, together with the name of the task we wish to halt. If we insert the line:

```
285 DISABLE mick
```

noting that this is within the declaration of 'fred', when 'fred' executes, 'mick' will be disabled, so that after 2 seconds have elapsed 'mick' will no longer be executed at all.
A task may also disable itself, so that once executed, it will never execute again - until re-enabled.
Consider:

```
125 DISABLE joe
```

As this is in the body of 'joe' this task is self-destructing !

We can, of course re-enable with ENABLE any disabled task at ANY

point in the program, even within the body of another task, remembering that a task which is disabled cannot execute, and therefore cannot enable itself.

```
100 TASK joe
110       B%=B%+1
120       PRINT TAB(5,7);" farewell - from joe !"
130       DISABLE joe
140 EXIT
```

The use of the wildcard '*' with DISABLE will disable ALL tasks

```
275 REM disable all tasks
276 DISABLE *
```

OMIT

The system is capable of supporting up to 8 'current' tasks, and a record is kept of these in a 'current' list. Note that an entry is made for each task declared at the start of a program, whether a task is enabled or not. It is possible that an application program may require to access more than 8 tasks, although no more than 8 may be current. The command OMIT, together with the task name, may be used to delete (and DISABLE) a task from the 'current' list, allowing for another task to be declared. We would suggest that such extra tasks required be declared at the END of the program, ideally defined as a BASIC PROCedure. Returning to our example program such an extra task could be declared as follows:

```
875 END
880 REM Extra tasks here
890 DEFPROCextra1
900 TASK joan
910       C%=C%+1
920       PRINT TAB(15,11);"joan says C% = ";C%;
930 EXIT
940 ENDPROC
```

We can OMIT, for example, task 'john' and enable the new task

'joan' in its place:

```
125 IF B% = 4 THEN OMIT john
126 IF B% = 4 THEN PROCextra1
127 IF B% = 4 ENABLE joan EVERY 100cs
```

You may if you wish remove ALL present current tasks with the use of the wildcard '*' after OMIT

```
125 IF B% = 4 THEN OMIT *
```

Controls on tasks declared and enabled

MULTI-BASIC's facilities allow tasks to declare and OMIT, and ENABLE and DISABLE, other tasks and themselves. With more complex applications written in MULTI-BASIC, the programmer may wish to determine, from program, which tasks are currently declared and enabled. For this purpose, the functions DECLD and ENABD are provided.

DECLD and ENABD

These are functions which return an integer value non-zero or zero. Examples of their use are:

```
40 IF DECLD["fred"] THEN GOTO 140
```

and:

```
70 IF ENABD["fred"] THEN PRINT "fred is running !"
```

so that a program may determine whether a given task is currently declared and/or enabled.

The single parameter of each of these is a task name delimited by "" (double quote) characters. If the wildcard character is used, as in:

```
90 IF DECLD["*"] THEN OMIT *
```

we can determine whether any task is declared. Similarly:

```
30 IF ENABD["*"] THEN PRINT "some task(s) running !"
```

allows us to check if any task is currently enabled. Of course a task cannot be enabled if it is not currently declared !

MULTI-BASIC maintains a list of all currently declared and enabled tasks. While the user refers to tasks by name, the system also has it's own task identities. These task identities are held as bit settings within a system-reserved byte, and the first task

to be declared is allocated bit-0, the 2nd task bit-1, and so on, with the 8th task allocated bit-7. It is this identity which is returned by DECLD and ENABD, so that:

```
50 PRINT DECLD["fred"]
```

would display either zero (if "fred" not currently declared) or 8 (if "fred" was the 4th task declared). The bit allocations and identities are:

bit	identity
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

Note that if OMIT has been used, and other tasks declared, then the identities may have been re-used.
The statement:

```
60 IF ENABD["fred"] = 16 THEN PRINT "currently 5th !"
```

will indicate if "fred" has task identity 16 (i.e. bit-5).

When the wildcard character is used, the functions return the full bit settings, eg:

```
80 PRINT DECLD["*"]
```

would display:

```
43
```

if tasks with bits 0, 1, 3, 5 were currently declared.

BBC Events

As we mentioned earlier, the BBC Microcomputer Operating System is constantly servicing interrupts from a variety of devices. The use of interrupts does give a certain transparency to various operations, for example entry of text via the keyboard whilst disc operations are performed. Note that you can 'type ahead' in many cases, and it appears that the machine is doing more than one thing at a time. This, of course is not true. Time is being divided between the various machine-code 'tasks' in the Operating System. Some of these routines can also be made to flag their discrete operation, and we call such flagging an 'event'. The Operating system can be made to give information about certain events via EVENTV, the event vector at &220, and these are fully documented in The Advanced User Guide.
Unfortunately, to handle such events correctly, the user must resort to machine-code, and in MULTI-BASIC, we do not use EVENTV at all, as the use of WHENEVER, with a suitably defined status, will allow BASIC users to control program flow in a more direct and easier implemented way. For those wishing to inteface via the EVNTV, an example program is given in the text.
If you refer to the entry for WHENEVER, you will note that the status must be defined as an integer variable, and that this comprises three distinct parts.

a) The two most significant bytes are the ADDRESS to test.

b) The next most significant byte is a MASK to apply to the result fetched from the address above.

c) The least significant byte is that(those) bit(s) you require INVERTed before the mask is applied.

With this mechanism, it is possible to examine ANY location in memory, every 1 centisecond, and either execute, or not, the specified task depending on the condition of the stated location.

As an example, suppose we wish to execute a task named 'jim' WHENEVER fire button '0' (on the Analogue connector) is pressed. Since this is connected to line 4 of PORTB (PB4) on the system

VIA mapped at &FE40, we can specify the address to test, and also which bit at that address ie:

a) The ADDRESS is &FE40 (PORTB).

b) MASK is &10 (line 4)

We do, however note that the active status of the line, that is when the button is pressed, will be 0 volts or in computer parlance TTL 'low', and therefore we must invert this for WHENEVER to do the job correctly ie:

c) INVERT is &10 (line 4)

We can now define the status variable as:

```
20 button0% = &FE401010
```

and instigate check of this status and conditional execution of 'jim' with:

```
30 ENABLE jim WHENEVER button0%
```

We can similarly test for the other button (connected to PB5) with:

```
40 button1% = &FE402020
```

or for execution conditional on either button with:

```
40 buttons% = &FE403030
```

A similar approach can be adopted in the case of user-supplied hardware connected to the User port. This is PORTB of the User VIA, and is mapped at &FE60. We could for example specify that all lines are to be inputs with:

```
10 uvia% = &FE60
20 CONFIGURE[DDRB;uvia%;%00000000]
```

and arrange to enable conditional execution on any of these lines being forced low. (They are held high with internal 'pullup' resistors)

There are two possible drawbacks to this approach.

1) that such switch closures are not automatically 'debounced', ie. the switch contacts will physically bounce open and close, giving spurious entries.
2) that the status is being checked every 1 centisecond, leading to possible multiple execution of the conditional task for as long as the status is valid.
We give the following program as an example of how to tackle the problem in a simple and economical way.

```
  10 REM Execution conditional on joystick 'FIRE' buttons
  20 REM System VIA at &FE40
  30 REM Button 0 on PB4
  40 REM Button 1 on PB5
  50 CLS
  60 REM preset button state indicators
  70 b0stat% = 0
  80 b1stat% = 0
  90 REM Define status for button 0 pressed
 100 b0_is_0% = &FE401010
 110 REM Define status for button 0 released
 120 b0_is_1% = &FE401000
 130 REM Define status for button 1 pressed
 140 b1_is_0% = &FE402020
 150 REM Define status for button 1 released
 160 b1_is_1% = &FE402000
 170 REM *************************
 180 REM Declare task to execute when Button 0 pressed/released
 190 TASK button0
 200 IF b0stat% = 1 THEN b0stat% = 0:ENABLE button0 WHENEVER
b0_is_0%:GOTO 230
 210       IF b0stat% = 0 THEN b0stat% = 1:ENABLE button0
WHENEVER b0_is_1%
 220       PRINTTAB(5,2);"Button 0 pressed ";:GOTO 240
 230       PRINTTAB(5,2);"Button 0 released";
```

```
  240 EXIT
  250 REM ************************
  260 REM Declare task to execute when Button 1 pressed/released
  270 TASK button1
  280 IF b1stat% = 1 THEN b1stat% = 0:ENABLE button1 WHENEVER
b1_is_0%:GOTO 310
  290   IF b1stat% = 0 THEN b1stat% = 1:ENABLE button1 WHENEVER
b1_is_1%
  300   PRINTTAB(5,3);"Button 1 pressed ";:GOTO 320
  310   PRINTTAB(5,3);"Button 1 released";
  320 EXIT
  330 REM ************************
  340 REM Enable conditional execution of above tasks
  350 ENABLE button0 WHENEVER b0_is_0%
  360 ENABLE button1 WHENEVER b1_is_0%
  370 REM Dummy 'foreground' ( The remainder of your program )
  380 REPEAT
  390 UNTIL 0
```

The two tasks are declared in an identical fashion, with each task taking action depending on what the previous status of its associated button was. In our case this is simply printing a report on the screen for each status change.

Other causes of 'events'.

Since the Operating System is responsible for the management of the input/output buffers (used for the keyboard, printer etc.), it is possible to conditionally execute BASIC tasks by checking the appropriate buffer 'busy' flags. These are physically located at &2CF-&2D7 in page2 of memory, and the following table indicates use.

Location	Use
&2CF	Keyboard buffer
&2D0	RS423 input buffer
&2D1	RS423 output buffer
&2D2	Printer buffer
&2D3-6	Sound channnel buffers 0-3
&2D7	Speech buffer

In each case, a buffer 'empty' is flagged by bit 7 in the associated location being set to 1, and this we can term a buffer-empty 'event'.

Printer spooler

The following example illustrates the use of task execution whenever the printer buffer is empty (bit 7 in &2D2 = 1).

The program is a printer spooler, and will copy any characters that are placed into a file, across to your printer. To make this more interesting, the foreground will accept display characters from the keyboard and place them into the file, giving the program a 'typewriter' function in which the typed text is left on file for later use. The program run is terminated when the backslash key is hit.
Note that unlike the previous example, it is unnecessary to check the location for both 1 or 0 in bit 7, as the Operating System automatically indicates the correct status at any particular time.

```
   10 REM ****************************
   20 REM  example of printer spooler
   50 REM ****************************
   60 REM
   70 REM first declare the task
   80 REM
   90 TASK spooler
  100    REM provide a trace
  110    x1%=r% MOD 16:r%=r%+1:x2%=r% MOD 16
  120    PRINT TAB(20+x2%,0);"*";
  130    PRINT TAB(20+x1%,0);" ";
  140    REM now the spooling logic
  150    REM save file pointer for foreground
  160    fptr% = PTR#file%
  170    REM exit if no more characters have been placed into
file
  180    REM i.e. if 'task' and 'foreground' file pointers are
equal
  190    REM n.b. also amend trace to indicate an early exit
  200    IF tptr%=fptr% THEN PRINT "!";
  210    IF tptr%=fptr% THEN EXIT
  220    REM redirect output to printer only
  230    *FX 3,10
  240    REM reset file pointer for task
  250    PTR#file% = tptr%
  260    REM transfer characters from file to printer, until EOF
is met
  270    REPEAT
  280       filech% = BGET#file%
  290       PRINT CHR$(filech%);
  300    UNTIL EOF#file%
  310    REM note updated file pointer, to be used next time task
executes
  320    tptr% = PTR#file%
  330    REM restore output to VDU
  340    *FX 3,0
  350    REM restore file pointer for foreground
  360    PTR#file% = fptr%
```

```
  370     REM finally exit from task
  380 EXIT
  390 REM
  400 REM clear VDU screen, display trace header, and preset
trace indicator
  410 REM
  420 CLS
  430 PRINT TAB(10,0);CHR$(129);
  440 PRINT "trace":PRINT:PRINT:PRINT
  450 r%=0
  460 REM
  470 REM open a file for output
  480 REM
  490 file% = OPENOUT("text")
  500 REM
  510 REM preset the copies of the file pointer which will be
used
  520 REM by foreground and task
  530 REM
  540 PTR#file% = 0
  550 fptr% = PTR#file%
  560 tptr% = fptr%
  570 REM
  580 REM now enable the task : the event which 'triggers' task
execution
  590 REM is an empty print buffer - this is indicated when
  600 REM the MS bit of location &2D2 is set
  610 REM
  620 empty% = &02D28000
  630 ENABLE spooler WHENEVER empty%
  640 REM
  650 REM and read characters from the keyboard
  660 REM n.b. that INKEY is used in order that foreground
execution continues
  670 REM
  680 REPEAT
  690     inch% = INKEY(100)
  700     IF inch%<>-1 THEN BPUT#file%,inch%
  710 UNTIL inch% = 92
  720 REM
```

```
730 REM hitting the backslash key will indicate end of input
740 REM
750 REM then disable spooler as no longer required
760 REM
770 DISABLE spooler
780 REM
790 REM and close file
800 REM
810 CLOSE#file%
820 STOP
```

The program could be augmented to provide a word-processing facility, in which the generated text if left in a nominated disc file, and automatically output to the printer.

Concurrent INPUT.

As we mentioned earlier, background tasks are slotted in between the foreground statements, and if we call a procedure which awaits keyed entry from the user, ALL program activity is suspended, until the called function is complete. One such example is that of INPUT, which gets text from the user until RETURN is pressed. This is clearly unsatisfactory, as the background tasks will halt for the duration of the function call. What is required is a function which behaves similarly to INPUT, but which allows all background tasks to run also. One method is to use the 'event' caused by a key press to build up an input string until RETURN is pressed, and the following example task utilises location &EC to flag such a key press and then add the key to a string. This location will contain the internal key number of the most recently pressed key, or zero if none is currently pressed. A complementary function FNinstring can then be called in a similar way to INPUT, this time without affecting operation of background tasks. We have set up a task to output the time to the screen every second, and demonstrate the concurrent function FNinstring accepting user commands to alter the Hours, Minutes and Seconds displayed.

```
 10 CLS
 20 REM define necessary variables
 30 ch%=-1:keys$="":string%=0
 40 A$=""
 50 REM Define event status variable
 60 kevnt% = &00ECFF00
 70 REM ****************************
 80 REM declare task to build string
 90 TASK keys
100  ch%=INKEY(0)
110  IF ch% = &0D THEN string%=1:DISABLE keys
120  IF ch%<>-1 THEN keys$=keys$+CHR$(ch%):PRINTCHR$(ch%);
130 EXIT
140 REM ****************************
150 REM Declare task to tell the time
160 TASK time
```

```
170       xp%=POS:yp%=VPOS
180       at%=@%
190       @%=0
200       time%=TIME
210       secs%=(time% DIV 100) MOD 60
220       mins%=(time% DIV 6000) MOD 60
230       hrs%=(time% DIV 360000) MOD 24
240       PRINTTAB(30,1);
250       IF hrs% < 10 THEN PRINT"0";
260       PRINThrs%;
270       PRINT":";
280       IF mins% < 10 THEN PRINT"0";
290       PRINTmins%;
300       PRINT":";
310       IF secs% < 10 THEN PRINT"0";
320       PRINTsecs%;
330       @%=at%
340       PRINTTAB(xp%,yp%);
350 EXIT
360 REM ***************************
370 ENABLE time EVERY 100cs
380 PRINTTAB(10,5);"1 Change Hours";
390 PRINTTAB(10,6);"2 Change Minutes";
400 PRINTTAB(10,7);"3 Change Seconds";
410 PRINTTAB(10,9);"Enter no. of choice";
420 PRINTTAB(12,11);">    ";TAB(13,11);
430  A$=FNinstring
440 choice% = EVAL(A$)
450 IF (choice% > 3 OR choice% < 1) THEN GOTO 420
460 PRINTTAB(10,9);
470 ON choice% GOTO 480,490,500
480 PRINT"Enter Hours          ";:GOTO 510
490 PRINT"Enter Minutes        ";:GOTO 510
500 PRINT"Enter Seconds        ";
510 PRINTTAB(13,11);"      ";TAB(13,11);
520 A$=FNinstring
530 value% = EVAL(A$)
540 rtime%=TIME:ON choice% GOTO 550,560,570
550 rtime%=rtime%-hrs%*360000+value%*360000:GOTO 580
560 rtime%=rtime%-mins%*6000+value%*6000:GOTO 580
```

```
570 rtime%=rtime%-secs%*100+value%*100
580 TIME=rtime%:GOTO 410
590 DEF FNinstring
600       ch%=-1:keys$="":string%=0
610       ENABLE keys WHENEVER kevnt%
620       REPEAT
630       UNTIL string%=1
640=keys$
650 END
```

Note that FNinstring is only called from the foreground (lines 370-580), and NOT from a task. If this were done, all tasks except 'keys' would halt for the duration of the string entry. Similar methods, can however be adopted for text entry to tasks and the following is one solution.

```
 10 CLS
 20 REM define necessary variables
 30 ch%=-1:keys$="":string%=0
 40 REM Define event status variable
 50 kevnt% = &00ECFF00
 60 REM ****************************
 70 REM declare task to build string
 80 TASK keys
 90  ch%=INKEY(0)
100  IF ch% = &0D THEN string%=1:DISABLE keys:GOTO 120
110  IF ch%<>-1 THEN keys$=keys$+CHR$(ch%):PRINTCHR$(ch%);
120 EXIT
130 REM ****************************
140 REM Declare task to tell the time
150 TASK time
160       xp%=POS:yp%=VPOS
170       at%=@%
180       @%=0
190       time%=TIME
200       secs%=(time% DIV 100) MOD 60
210       mins%=(time% DIV 6000) MOD 60
220       hrs%=(time% DIV 360000) MOD 24
230       PRINTTAB(30,1);
240       IF hrs% < 10 THEN PRINT"0";
```

```
250       PRINThrs%;
260       PRINT":";
270       IF mins% < 10 THEN PRINT"0";
280       PRINTmins%;
290       PRINT":";
300       IF secs% < 10 THEN PRINT"0";
310       PRINTsecs%;
320       @%=at%
330       PRINTTAB(xp%,yp%);
340 EXIT
350 REM ****************************
360 REM Declare task to test for CLOCKON or OFF
370 TASK test
380       IF ENABD["keys"]=0 THEN ENABLE keys WHENEVER kevnt%
390       IF string%=0 THEN EXIT
400       IF keys$ = "CLOCKOFF" THEN DISABLE time:GOTO 420
410       IF keys$ = "CLOCKON" THEN ENABLE time EVERY 100cs
420       string%=0:keys$="":PRINT
430 EXIT
440 ENABLE time EVERY 100cs
450 ENABLE test EVERY 25cs
460 REM DUMMY FOREGROUND
470 PRINTTAB(0,3);
480 REPEAT
490 UNTIL 0
```

Note that in all examples of keyed entry while a task is running that changes the TAB positions, it is not possible to have the input and output cursors seperate. (ie. with the use of the cursor keys and COPY) Since these are generally used for editing, this is not really a problem.

Use of EVNTV

As already noted in the section on BBC events, the use of WHENEVER, with a suitable defined status variable, will usually suffice for the majority of applications. For those wishing to use EVENTV, we need a short section of machine-code to interface with MULTI-BASIC. The example below is similar to that given in The Advanced Guide, except instead of giving the sound command in machine-code we've chosen to do this in a task. The key value is passed in the low byte of the resident integer A%, and bit 7 of the next-most significant byte of A% is set to 1 when the event occurs. We simply enable execution of the task with WHENEVER.

```
 10 DIM CODE 100
 20 EVENTV = &220
 30 FOR opt% = 0 TO 3 STEP 3
 40 P%=CODE
 50 [ OPT opt%
 60 PHP:PHA:CMP #2:BNE xit  was it our event ?
 70 STY &404  put key val in lo byte of A%
 80 LDA #&80:STA &405 flag event to MULTI-BASIC
 90 .xit PLA:PLP:RTS
100]
110 NEXT
120 ?EVENTV = CODE MOD 256
130 EVENTV?1 = CODE DIV 256
140 REM **************************
150 REM task to give sound when key pressed
160 TASK ksound
170       pitch%=A%
180       A%=0:REM clear status
190       SOUND 1,-15,pitch%,1
200 EXIT
210 kevnt% = &4058000
220 ENABLE ksound WHENEVER kevnt%
221 REM enable OS event 2
230 *FX14,2
240 REM Dummy foreground
250 REPEAT:s%=INKEY(0):UNTIL0
```

*EVENTSON & *EVENTSOFF

These two commands provide a convenient means of enabling and disabling the interrupts used by MULTI-BASIC. For any multi-tasking to work, these interrupts MUST be enabled with *EVENTSON if previously disabled with *EVENTSOFF.
After initialising at switch-on, the default state of the interrupts is on, ie as if you had given the command *EVENTSON. There is therfore no need to enable these on first initialising MULTI-BASIC.
You should disable these interrupts before initialising another language, as locations in zero page will be affected. Note that these two commands are only available whilst MULTI-BASIC is the currently initialised language ROM.

Since it is essential that no event causes execution of any previously declared tasks whilst loading in a new program, the interrupts should be disabled if you wish to CHAIN programs. This is not required for normal load and save operations, or for random disc access. Below is given an example of the use of *EVENTSON and *EVENTSOFF with CHAIN.
The first program will CHAIN the second from the task 'green' when this is executed. Note that there is no need in this case for *EVENTSOFF, as no event is allowed to cause transfer of control from a currently executing task.
In the second program, however, the first one is CHAINed from the foreground, so in this case we use *EVENTSOFF to halt all current events.

```
  1 *EVENTSON
 10 REM ****************************
 20 REM     example of chaining
 30 REM ****************************
 40 TASK red
 50    x1%=r% MOD 16:r%=r%+1:x2%=r% MOD 16
 60    PRINT TAB(20+x2%,0);"*";
 70    PRINT TAB(20+x1%,0);" ";
 80 EXIT
 90 TASK green
100    x1%=g% MOD 16:g%=g%+1:x2%=g% MOD 16
110    PRINT TAB(20+x2%,1);"*";
120    PRINT TAB(20+x1%,1);" ";
121    CHAIN"chained"
130 EXIT
131 r%=0:g%=0
240 ENABLE red    EVERY  25cs
250 ENABLE green  EVERY  50cs
280 MODE 7
300 PRINT TAB(10,0);CHR$(129);"red     :";
310 PRINT TAB(10,1);CHR$(130);"green   :";
380 REPEAT
390 UNTIL 0
400 STOP
```

```
  5 *EVENTSON
 10 REM ****************************
 20 REM     example of chaining (2)
 30 REM ****************************
 40 TASK red
 50    x1%=r% MOD 16:r%=r%+1:x2%=r% MOD 16
 60    PRINT TAB(20+x2%,0);"*";
 70    PRINT TAB(20+x1%,0);" ";
 80 EXIT
 90 TASK green
```

```
100     x1%=g% MOD 16:g%=g%+1:x2%=g% MOD 16
110     PRINT TAB(20+x2%,1);"*";
120     PRINT TAB(20+x1%,1);" ";
130 EXIT
240 ENABLE red     EVERY  25cs
250 ENABLE green   EVERY  50cs
280 MODE 7
290 r%=0:g%=0
300 PRINT TAB(10,0);CHR$(129);"red     :";
310 PRINT TAB(10,1);CHR$(130);"green   :";
380 FOR X = 1 TO 1200
381 PRINT TAB(20,20);"#####";
390 NEXT
391 *EVENTSOFF
400 CHAIN"chain"
```

Tackling the unexpected

Task overrun

The mechanism of the multi-tasking is such that when a task interval 'times-out', or an event occurs DURING the execution of a background task, the new task (which cannot execute immediately) is added to a list of 'asserted' tasks, which will be executed at the earliest opportunity i.e. in the next available time slot. A situation may arise, usually as a result of excessive task length, that a task will not be executed at the correct time-interval, but is always late - or is almost permanently executing, to the detriment of the foreground task. The user should attempt to avoid such situations, and also be aware as to their cause and cure. We have called such a phenomenon a task 'overrun', and fortunately there is an easy way of detecting the occurence of such 'overruns'.

Overrun Detection

Using the: ENABLE .... WHENEVER construct, we can engender the execution of a task when an overrun occurs, by checking the status of the OS variable holding any tasks which are so asserted, i.e. if no overruns occur, the contents of this variable will be zero, otherwise it will contain a bit-map of the tasks 'overrun' by system timing, program execution, or an asynchronous event. We have given the name 'assert' to the variable, and it comprises 1 byte at location &66 in zero-page memory. The user is invited to examine this location as often as he/she sees fit, but NEVER WRITE TO THIS LOCATION. If you do, it is possible the operating system will crash. We can specify to test this location as:

```
760 over% = &0066FF00:REM bit-map checks ALL for test
770 REM check if any tasks 'left over'
780       ENABLE check WHENEVER over%
```

and define a task 'check' elsewhere in the program to give a suitable report.
The OS variable 'assert' is bit-mapped as follows, where a bit set to 1 represents an overrun condition on the associated task.

bit no.	task no.
7	8
6	7
5	6
4	5
3	4
2	3
1	2
0	1

Remembering that task 1 is the first task defined in the program etc.

Cause of overrun

1. Multiple task execution

As the system expects to be able to service ALL timed-out, or valid asynchronous tasks in one time-slot stolen from the foreground, the total time required is the sum of the execution times of all such 'live' tasks - i.e all that are required to be executed. For the purposes of this discussion, let us assume we have defined 8 tasks, and they at some point in time, will ALL be executed in the time-slot. Also, that the total time required to execute them all is 25 centi-seconds. If one of these tasks has been enabled for EVERY 10 centiseconds, then this task will overrun and assert itself at the wrong time-intervals - remember each task is indivisible and cannot be interrupted by another. Quite clearly, the program will not perform as the user expects, and an examination and re-structuring of the program will be necessary.

2. Events

The second, but less serious cause of overrun, is where an enabled event has occurred. In this case, it is very likely we will wish the task asserted as soon as possible. This then, is not really an error condition, and requires no correction.

Overrun exceptions

All newly timed-out or event-generated tasks are allowed to assert an overrun condition. What is NOT allowed is an overrun condition to develop on a task which is currently executing, or marked for execution in the current time-slot.
The reason for this is that a task may be required to clear the status of a register actually causing an 'event'. Due to the discrete time taken for transfer of control to the task, and the overheads in parsing BASIC, a check of the status may occur again before the status is cleared, causing multiple execution of the task.

Unserviced 'events'

As an example of what will occur if you omit to service an event correctly, type in and the RUN the program given below.
We have set up Timer 2 of the User VIA to time out every 5 centiseconds. Note that we have disabled the production of an interrupt, as these cannot be serviced from BASIC. The VIA Interrupt Flag Register will, however, reflect the status of the timer with bit 5 being set to 1 when the timer times out, and this status is checked every 1 centisecond due to the WHENEVER statement.
The enabled task increments, then prints on the VDU the integer A%. It also re-writes timer 2 hi-counter to clear the associated bit in the Interrupt Flag Register, and restart the count-down When RUN, every time Timer 2 times out, the task will be executed, and you will see that this is at 5cs intervals.

Now try putting a REM on line 100, and RUN the program again. The result is clearly not what we intended, with the task asserting itself at the earliest possible moment. It is a BASIC version of what can happen at a machine-code level, when an interrupting condition is allowed to assert itself repeatedly and the users code totally disregards acknowledging the interrupt !

We cannot stress heavily enough the importance of servicing ALL possible enabled events correctly, and where use of the VIAs are concerned, complete understanding of what is required is absolutely essential. The user is referred to the 6522 documentation for further information.

```
10 CLS
20 A%=0
30 REM Task executed on time out of VIA timer 2
40 REM User VIA at &FE60
50 uvia%=&FE60
60 REM **********************
70 TASK timer
```

```
 80       A%=A%+1
 90       PRINTTAB(5,5);A%
100       CONFIGURE[T2C-H;uvia%;&C3]
110 EXIT
111 REM ************************
120 REM set up T2 for 1 shot timing
130 REM Disable T2 interrupts
140 CONFIGURE[VIER;uvia%;%00100000]
141 REM &C350 will give 5cs intervals
150 REM Load up T2 lo latch
160 CONFIGURE[T2C-L;uvia%;&50]
161 REM Load up T2 hi latch/start count
170 CONFIGURE[T2C-H;uvia%;&C3]
171 REM define event status
180 tevnt%=&FE6D2000
190 ENABLE timer WHENEVER tevnt%
191 REM Dummy foreground
200 REPEAT:UNTIL0
```

Escape & BREAK

Whilst developing your program, you may notice the odd occaision when pressing the Escape key has no effect. For reasons too involved to discuss here, this is an unfortunate by-product of running BASIC programs under interrupt. In this event hit BREAK, and retrieve your program with OLD.

Section 2. Keywords

Index to this section

1. CONFIGURE
4. DECLD
6. DISABLE
7. ENABLE
10. ENABD
12. EVERY
13. EXIT
14. FETCH
17. OMIT
18. RD_TIMER
19. SETBITS
22. SET_TIMER
23. SHIFTL
25. SHIFTR
27. TGGLE
30. UNSET
33. WAIT
34. WHENEVER

Section 2. Keywords

****

CONFIGURE

****

Purpose

To configure a 6522 VIA register by assigning to it a specified bit-pattern.

Syntax

CONFIGURE[<register name>;<device address>;<bit pattern>]

Parameters

Parameter 1. <register name>

This can be any of the following:

parameter name	description
PORTA	ORA/IRA output/input register
PORTB	ORB/IRB output/input register
DDRA	Data-Direction for PORTA
DDRB	Data-Direction for PORTB
T1C-L	Timer 1 low-order counter
T1C-H	Timer 1 high-order counter
T1L-L	Timer 1 low-order latch
T1L-H	Timer 1 high-order latch
T2C-L	Timer 2 low-order latch/counter
T2C-H	Timer 2 high-order counter
SR	Shift Register
ACR	Auxiliary Control Register
PCR	Peripheral Control Register
VIER	Interrupt Enable Register
VIFR	Interrupt Flag Register
PTANH	As PORTA but no handshake

Parameter 2. <device address>

This may be given in either of two forms thus:

a) As a literal hex. string prefixed with '&', e.g:

```
&FE60
```

b) As an integer variable which contains the address. In this case the variable MUST have been pre-defined and initialised correctly, e.g:

```
pvia% = &FE60
```

Parameter 3. <bit pattern>

This may be given in one of three forms thus:

a) As a literal Binary string prefixed with '%', e.g:

```
%01010101
```

noting that between 1 and 8 bits may be quoted i.e:

```
%1
```

will be assumed to be bit 0.

b) As a literal hex. string prefixed with '&', e.g:

```
&55
```

c) As the contents of a previously defined integer variable, e.g:

```
direction% = &55
```

Note that in this case only the least-significant byte of the variable is used.

Example

The following sequence could be used to configure Port B of the user VIA mapped at &FE60:

```
   10 pvia% = &FE60 :REM Base address of Device
   20 CONFIGURE[DDRB;pvia%;%00001111]
:REM 0-3 Output, 4-7 input
```

Associated keywords

SETBITS, UNSET, TGGLE, SHIFTL, SHIFTR

****

DECLD

****

Purpose

A function which indicates whether a specified task, or any tasks, are currently declared. The function returns an integer value non-zero or zero.

If the specified task is not declared, then the value returned will be zero. If the task is currently declared, then the value returned will be the system's task identity : this will be one of the bits in the LS byte of the integer. See the section on multi-tasking for further details.

Syntax

DECLD["<task name>"]

or

DECLD["*"]

Parameters

Parameter 1. <task name>

This may consist of between 1 and 7 display characters (ASCII codes &21-&7E), with the exception of "" (ASCII code 34) - which delimits the name. The name may not consist of the wildcard character '*' only. Use of the wildcard character indicates any task(s).

Example

The following examples illustrate typical use :

```
60 IF DECLD["timer"] THEN OMIT timer
```

and

```
80 IF DECLD["*"] THEN OMIT *
```

Additionally

```
160 PRINT DECLD["timer"]
```

will cause zero or an integer value to be displayed.

Associated keywords

TASK, EXIT, ENABLE, OMIT, EVERY, WHENEVER, DISABLE, ENABD

****

DISABLE

****

Purpose

Used to cease the operation of a selected concurrent 'task'.

Syntax

DISABLE<spc><task.name>

Operation

The selected task is deleted from the current list of active background tasks, until re-activated with ENABLE. Note that the command has a distinctly different action to that of OMIT. Use of the wildcard character '*' will disable all tasks.

Examples

```
90 IF count > 30 THEN DISABLE keys
```

will (conditionally) disable task 'keys', and

```
70 DISABLE *
```

will disable all current tasks.

Associated Keywords

ENABLE, TASK, EXIT, OMIT, EVERY, WHENEVER, DECLD, ENABD

****

ENABLE

****

Purpose

To instigate the operation of a concurrent 'task'. The task must have been previously declared by executing the TASK.....EXIT construct, otherwise an error message will be given. The command may be given in two different forms, depending on whether the task is to be executed at regular intervals, or on some asynchronous 'event'.

Syntax

a) ENABLE<spc><task.name><spc>EVERY<spc><time.interval><cs>

or:

b) ENABLE<spc><task.name><spc>WHENEVER<spc><status>

Operation:

The previously defined <task.name> is executed whenever the interval <time> is achieved or exceeded, or when a specified status condition exists.

Parameters

Parameter 1. (options a & b) <task.name>

This may consist of between 1 and 7 display characters (ASCII codes &21-&7E), with the exception of "" (ASCII code &22).

Parameter 2. (option a) <time.interval>

This must be given as a literal decimal string, e.g:

```
100
```

The value given will be stated in centiseconds, and may be in the range 0-65535 (unsigned 16-bit integer).
The term 'cs' is a literal string.

Parameter 2. (option b) <status>

This must be given as a previously assigned-to integer variable, e.g:

```
evnt%=&FE6D0200
```

and the integer will be treated as three distinct parts by the operating system:

```
<address><mask><invert>
```

<address> is the two-byte address from which the status will be read, <mask> an 8-bit mask specifying the bit or bits to test, and <invert> an 8-bit value specifying which bits in the retrieved status require inverting, before the mask is applied.

Example

```
  5 REM Define a task to read a port
 10 TASK keys
 20         FETCH[PORTA;svia%;mask%;result%]
 30 EXIT
 40 REM More program here
 50 REM Enable the task "keys" to read the port every second
 60 ENABLE keys EVERY 100cs
 65 REM Enable the task "keys" if the IFR register at &FE6D =
02
 70 evnt% =&FE6D0200
 80 ENABLE keys WHENEVER evnt%
```

Associated keywords

DISABLE, OMIT, EVERY, TASK, EXIT, WHENEVER, DECLD, ENABD

****

ENABD

****

Purpose

A function which indicates whether a specified task, or any tasks, are currently enabled i.e. executing on EVERY or WHENEVER conditions. The function returns an integer value non-zero or zero.

If the specified task is not enabled, then the value returned will be zero. If the task is currently enabled, then the value returned will be the system's task identity : this will be one of the bits in the LS byte of the integer. See the section on multi-tasking for further details.

Syntax

ENABD["<task name>"]

or

ENABD["*"]

Parameters

Parameter 1. <task name>

This may consist of between 1 and 7 display characters (ASCII codes &21-&7E), with the exception of '"' (ASCII code 34) - which delimits the task name. The name may not consist of the wildcard character '*' only. Use of the wildcard character indicates any task(s).

Example

The following examples illustrate typical use :

```
20 IF ENABD["timer"] THEN PRINT "timer running !"
```

and

```
80 IF ENABD["*"] THEN PRINT 'running !"
```

Additionally

```
160 PRINT ENABD["timer"]
```

will cause zero or an integer value to be displayed.

Associated keywords

TASK, EXIT, ENABLE, OMIT, DISABLE, EVERY,WHENEVER, DECLD

****

EVERY

****

Purpose

A syntactic link word used in conjunction with ENABLE, which is followed by the time interval required for concurrent task execution.

Syntax

ENABLE<spc><task.name><spc>EVERY<spc><time.interval><cs>

Example

```
120 ENABLE toggle EVERY 50cs
```

Associated keywords

ENABLE, DISABLE, TASK, EXIT, OMIT, EVERY, DECLD, ENABD

****

EXIT

****

Purpose

Mark the end of the declaration of a particular concurrent task. Each definition of a task must always end in EXIT, as DEFPROC in ended in ENDPROC. The EXIT delimiting a task must occur in a statement alone, as in the 1st example below. Note, however, that EXIT may be included in a conditional statement, as in the 2nd example below.

Syntax

TASK<name><executable statement(s)>EXIT

Examples

```
100 TASK toggle
110          TGGLE[PORTB;motors%;%00000001]
120 EXIT
```

and

```
50 TASK invert
60 :
70    IF total > 50 THEN EXIT
80 :
90 EXIT
```

Associated Keywords

TASK, DISABLE, ENABLE, EVERY, WHENEVER, OMIT, DECLD, ENABD

****

FETCH

****

Purpose

To read the contents of a specified VIA register, and after logically ANDing with a given bit-pattern, write the result to a specified location in memory.

Syntax

FETCH[<register name><device address>;<bit pattern>;
<destination>]

Parameters

Parameter 1. <register name>

This can be any of the following:

parameter name	description
PORTA	ORA/IRA output/input register
PORTB	ORB/IRB output/input register
DDRA	Data-Direction for PORTA
DDRB	Data-Direction for PORTB
T1C-L	Timer 1 low-order counter
T1C-H	Timer 1 high-order counter
T1L-L	Timer 1 low-order latch
T1L-H	Timer 1 high-order latch
T2C-L	Timer 2 low-order latch/counter
T2C-H	Timer 2 high-order counter
SR	Shift Register
ACR	Auxiliary Control Register
PCR	Peripheral Control Register
VIER	Interrupt Enable Register
VIFR	Interrupt Flag Register
PTANH	As PORTA but no handshake

Parameter 2. <device address>

This may be given in either of two forms thus:

a) As a literal Hex. string prefixed with '&', e.g:

```
&FE60
```

b) As an integer variable which contains the address. In this case the variable MUST have been pre-defined and initialised correctly, e.g:

```
svia% = &FE60
```

Parameter 3. <bit pattern>

This may be given in one of three forms thus:

a) As a literal Binary string prefixed with '%', e.g:

```
%11110000
```

b) As a literal Hex. string prefixed with '&', e.g:

```
&F0
```

c) As the contents of a previously defined integer variable, e.g:

```
mask% = &F0
```

Note that in this case only the least-significant byte of the variable is used.

Parameter 4. <destination>

This may be given in one of two forms:

a) As a literal Hex. address in memory, e.g:

&2400

b) As an integer variable to accept the result, e.g:

result%

Note that in this case if "result%" already exists i.e has been previously defined, this will be used, otherwise the variable will be created. After executing the procedure which uses such a Basic integer, the specified integer will contain the result in its least-significant byte, the three most significant bytes being zeroised.

Example

```
80 FETCH[PORTA;svia%;%11110000;result%]
```

Associated keywords

CONFIGURE, SETBITS, TGGLE, SHIFTR, SHIFTL, UNSET

****

OMIT

****

Purpose

To disable and then delete a quoted 'task' from the current list, thus allowing the inclusion of another.

Syntax

OMIT<task.name>

If the task.name is given with the wildcard '*', ALL currently resident tasks will be removed from the 'current' list, making room for a further eight.

Examples

```
120 OMIT do_time

500 OMIT *
```

Associated keywords

DISABLE, ENABLE, TASK, EXIT, EVERY, WHENEVER, DECLD, ENABD

****

RD_TIMER

****

Purpose

To read the current 4 byte value of a selected system timer. Each system timer is incremented every 1 centisecond (when interrupts are enabled).

Syntax

RD_TIMER[<timer.number>;<result>]

Parameters

Parameter 1. <timer.number>

This should be quoted as a literal Decimal number in the range 1-5.

Parameter 2. <result>

An integer variable is required for this, and if the variable quoted does not exist, then it will be created.

Example

```
20 RD_TIMER[3;time3%]
```

Associated keyword

SET_TIMER

****

SETBITS

****

Purpose

To set a specific bit or bits to 1 in a selected VIA register.

Syntax

SETBITS[<register.name>;<device.address>;<bit.pattern>]

Parameters

Parameter 1. <register name>

This can be any of the following:

parameter name	description
PORTA	ORA/IRA output/input register
PORTB	ORB/IRB output/input register
DDRA	Data-Direction for PORTA
DDRB	Data-Direction for PORTB
T1C-L	Timer 1 low-order counter
T1C-H	Timer 1 high-order counter
T1L-L	Timer 1 low-order latch
T1L-H	Timer 1 high-order latch
T2C-L	Timer 2 low-order latch/counter
T2C-H	Timer 2 high-order counter
SR	Shift Register
ACR	Auxiliary Control Register
PCR	Peripheral Control Register
VIER	Interrupt Enable Register
VIFR	Interrupt Flag Register
PTANH	As PORTA but no handshake

Parameter 2. <device.address>

This may be given in either of two forms thus:

a) As a literal Hex. string prefixed with '&', e.g:

```
&FE60
```

b) As an integer variable which contains the address. In this case the variable MUST have been pre-defined and initialised correctly, e.g:

```
pvia% = &FE60
```

Parameter 3. <bit-pattern>

Only those bits given as '1' will be set, other bits being unaffected.The pattern may be given in one of three forms thus:

a) As a literal Binary string prefixed with '%', e.g:

```
%01010101
```

b) As a literal Hex. string prefixed with '&', e.g:

```
&55
```

c) As the contents of a previously defined integer variable, e.g:

```
setbits% = &55
```

Note that in this case only the least-significant byte of the variable is used.

Example

```
10 bits% = &11
20 via%  = &FE60
30 SETBITS[PORTA;via%;bits%]
```

Associated keywords

UNSET, TGGLE, FETCH, CONFIGURE, SHIFTL, SHIFTR

****

SET_TIMER

****

Purpose

To write a new 4 byte value to a selected system timer. Each system timer is incremented every 1 centisecond (when interrupts are enabled).

Syntax

SET_TIMER[<timer.number>;<value>]

Parameters

Parameter 1. <timer.number>

This should be quoted as a literal Decimal number in the range 1-5.

Parameter 2. <value>

This should be a 4 byte value, and must be given as the contents of a previously defined integer variable, e.g:

```
interval% = -100
```

The value assigned may be in the range -2147483648 to +2147483647 inclusive.

Example

```
20 SET_TIMER[3;interval%]
```

Associated keyword

RD_TIMER

****

SHIFTL

****

Purpose

To shift left (multiply by 2) the contents of a specified register on a 6522 VIA.

Syntax

SHIFTL[<register.name>;<device.address>]

Parameters

Parameter 1. <register name>

This can be any of the following:

parameter name	description
PORTA	ORA/IRA output/input register
PORTB	ORB/IRB output/input register
DDRA	Data-Direction for PORTA
DDRB	Data-Direction for PORTB
T1C-L	Timer 1 low-order counter
T1C-H	Timer 1 high-order counter
T1L-L	Timer 1 low-order latch
T1L-H	Timer 1 high-order latch
T2C-L	Timer 2 low-order latch/counter
T2C-H	Timer 2 high-order counter
SR	Shift Register
ACR	Auxiliary Control Register
PCR	Peripheral Control Register
VIER	Interrupt Enable Register
VIFR	Interrupt Flag Register
PTANH	As PORTA but no handshake

Parameter 2. <device.address>

This may be given in either of two forms thus:

a) As a literal Hex. string prefixed with '&', e.g:

```
&FE60
```

b) As an integer variable which contains the address. In this case the variable MUST have been pre-defined and initialised correctly, e.g:

```
via% = &FE60
```

Example

```
120 via% = &FE60
130 SHIFTL[PORTB;via%]
```

Associated keywords

SHIFTR, CONFIGURE, SETBITS, UNSET, TGGLE, FETCH

****

SHIFTR

****

Purpose

To shift right (divide by 2) the contents of a specified register on a 6522 VIA.

Syntax

SHIFTR[<register.name>;<device.address>]

Parameters

Parameter 1. <register name>

This can be any of the following:

parameter name	description
PORTA	ORA/IRA output/input register
PORTB	ORB/IRB output/input register
DDRA	Data-Direction for PORTA
DDRB	Data-Direction for PORTB
T1C-L	Timer 1 low-order counter
T1C-H	Timer 1 high-order counter
T1L-L	Timer 1 low-order latch
T1L-H	Timer 1 high-order latch
T2C-L	Timer 2 low-order latch/counter
T2C-H	Timer 2 high-order counter
SR	Shift Register
ACR	Auxiliary Control Register
PCR	Peripheral Control Register
VIER	Interrupt Enable Register
VIFR	Interrupt Flag Register
PTANH	As PORTA but no handshake

Parameter 2. <device.address>

This may be given in either of two forms thus:

a) As a literal Hex. string prefixed with '&', e.g:

```
&FE60
```

b) As an integer variable which contains the address. In this case the variable MUST have been pre-defined and initialised correctly, e.g:

```
svia% = &FE60
```

Example

```
120 svia% = &FE60
130 SHIFTR[PORTB;via%]
```

Associated keywords

SHIFTL, CONFIGURE, SETBITS, UNSET, TGGLE, FETCH

****

TGGLE

****

Purpose

To perform an Exclusive-OR on a specific bit or bits in a selected VIA register.

Syntax

TGGLE[<register.name>;<device.address>;<bit.pattern>]

Parameters

Parameter 1. <register name>

This can be any of the following:

parameter name	description
PORTA	ORA/IRA output/input register
PORTB	ORB/IRB output/input register
DDRA	Data-Direction for PORTA
DDRB	Data-Direction for PORTB
T1C-L	Timer 1 low-order counter
T1C-H	Timer 1 high-order counter
T1L-L	Timer 1 low-order latch
T1L-H	Timer 1 high-order latch
T2C-L	Timer 2 low-order latch/counter
T2C-H	Timer 2 high-order counter
SR	Shift Register
ACR	Auxiliary Control Register
PCR	Peripheral Control Register
VIER	Interrupt Enable Register
VIFR	Interrupt Flag Register
PTANH	As PORTA but no handshake

Parameter 2. <device.address>

This may be given in either of two forms thus:

a) As a literal Hex. string prefixed with '&', e.g:

```
&FE60
```

b) As an integer variable which contains the address. In this case the variable MUST have been pre-defined and initialised correctly, e.g:

```
cvia% = &FE60
```

Parameter 3. <bit-pattern>

Only those bits given as '1' will be affected. The pattern may be given in one of three forms thus:

a) As a literal Binary string prefixed with '%', e.g:

```
%11001100
```

b) As a literal Hex. string prefixed with '&', e.g:

```
&CC
```

c) As the contents of a previously defined integer variable, e.g:

```
togbits% = &CC
```

Note that in this case only the least-significant byte of the variable is used.

Example

```
10 togbits% = &CC
20 svia%   = &FE60
30 TGGLE[PORTA;svia%;bits%]
```

Associated keywords

SETBITS, UNSET, FETCH, CONFIGURE, SHIFTR, SHIFTL

****

UNSET

****

Purpose

To clear a specific bit or bits to 0 in a selected VIA register.

Syntax

UNSET[<register.name>;<device.address>;<bit.pattern>]

Parameters

Parameter 1. <register name>

This can be any of the following:

parameter name	description
PORTA	ORA/IRA output/input register
PORTB	ORB/IRB output/input register
DDRA	Data-Direction for PORTA
DDRB	Data-Direction for PORTB
T1C-L	Timer 1 low-order counter
T1C-H	Timer 1 high-order counter
T1L-L	Timer 1 low-order latch
T1L-H	Timer 1 high-order latch
T2C-L	Timer 2 low-order latch/counter
T2C-H	Timer 2 high-order counter
SR	Shift Register
ACR	Auxiliary Control Register
PCR	Peripheral Control Register
VIER	Interrupt Enable Register
VIFR	Interrupt Flag Register
PTANH	As PORTA but no handshake

Parameter 2. <device.address>

This may be given in either of two forms thus:

a) As a literal Hex. string prefixed with '&', e.g:

```
&FE60
```

b) As an integer variable which contains the address. In this case the variable MUST have been pre-defined and initialised correctly, e.g:

```
device% = &FE60
```

Parameter 3. <bit-pattern>

Only those bits given as '1' will be cleared, other bits being unaffected. The pattern may be given in one of three forms thus:

a) As a literal Binary string prefixed with '%', e.g:

```
%00001111
```

b) As a literal Hex. string prefixed with '&', e.g:

```
&0F
```

c) As the contents of a previously defined integer variable, e.g:

```
clrbits% = 15
```

Note that in this case only the least-significant byte of the variable is used.

Example

```
10 clrbits% = &0F
20 dev%   = &FE60
30 UNSET[PORTB;dev%;clrbits%]
```

Associated keywords

SETBITS, TGGLE, FETCH, CONFIGURE, SHIFTR, SHIFTL

****

WAIT

****

Purpose

Suspend all program activity for a specified interval.

Syntax

WAIT[<millisecs>]

Parameters

Only one parameter is required, and this is the desired delay in milliseconds. It may be quoted in one of two forms thus:

a) As a literal Hex value in the range &0 to &FFFF, e.g:

```
&64
```

b) As the contents of a previously defined integer variable, e.g:

```
delay% = 100
```

In this case only the two LS bytes of the variable will be used.

Examples

```
40 WAIT[&64] :REM wait 100 millisecs

40 delay% = 1000
50 WAIT[delay%] :REM wait 1 second
```

****

WHENEVER

****

Purpose

A syntactic link word used in conjunction with ENABLE, which is followed by an integer variable specifying an 'event' condition.

Syntax

ENABLE<spc><task.name><spc>WHENEVER<spc><status>

<status> MUST be given as a previously assigned-to integer variable, and this comprises 3 distinct parts reading left to right (MSB to LSB):

<address><mask><invert>

<address> is the address from which a status byte will be read.
<mask> is an 8-bit value corresponding to the bit or bits to test.
<invert> is an 8-bit value specifying which bits of the status to invert before applying the mask.

Thus the integer is made up as <address> in its most significant bytes, and <mask><invert> in the least signicant part.

Example

```
100 REM Define a variable to test bit 1 of address &FE6D
110 evnt% = &FE6D0200
120 ENABLE keys WHENEVER evnt%
```

Associated keywords

ENABLE, DISABLE, TASK, EXIT, OMIT, EVERY, DECLD, ENABD

Section 3. Versatile Interface Adaptor Control

Index to this section.

1. Procedure names and syntax.
3. Parameter values.
    Register
4. Device Addresses
    Bit-pattern
5. Location
6. Summary of parameter types.
8. Indirection operators.
9. Example procedure calls.

Section 3. VIA Control

VIA Control Words

Seven procedures are provided for Read/Write of the BBC machines Versatile Interface Adaptors (VIAs). These are summarised below together with their respective actions:

procedure	action
CONFIGURE	configure the device/register by assigning a specified bit-pattern
SETBITS	set bits in device/register
UNSET	unset bits in a device/register
TGGLE	toggle bits in device/register
SHIFTL	shift device register left
SHIFTR	shift device register right
FETCH	get bits from device register, logically AND with given bit-pattern, then place into specified location

The syntax for the various procedures is as follows:

CONFIGURE[<register>;<device address>;<bit-pattern>]

SET[<register>;<device address>;<bit-pattern>]

UNSET[<register>;<device address>;<bit-pattern>]

TGGLE[<register>;<device address>;<bit-pattern>]

SHIFTL[<register>;<device address>]

SHIFTR[<register>;<device address>]

FETCH[<register>;<device address>;
<bit-pattern>;<location>]

Example use

A typical procedure call using CONFIGURE to write to the Auxilliary Control Register would be:

```
CONFIGURE[ACR;tempr%;%01110101]
```

and the parameters in this example are:

parameter	meaning
ACR	specifies the ACR register
tempr%	contains the base address of device
%01110101	indicates bit settings

CONFIGURE, SET, UNSET and TGGLE have these three 'equivalent' parameters, while SHIFTL and SHIFTR have only the first two of these parameters. FETCH has an additional parameter, indicating a location for the FETCHed bits.

A typical call using FETCH could be:

```
FETCH[IFR;tempr%;%00001001;bits03%]
```

where the 4th parameter is :

parameter	meaning
bits03%	variable to receive bits 0 and 3 of the IFR register

Parameter values

Parameters may be specified in the forms:

1. Register

<Register> may be specified by one of the following names:

register	no.	R6522 designation
PORTB	0	ORB/IRB output/input register B
PORTA	1	ORA/IRA output/input register A
DDRB	2	DDRB data direction register B
DDRA	3	DDRA data direction register A
T1C-L	4	T1 low-order latches/counter
T1C-H	5	T1 high-order counter
T1L-L	6	T1 low-order latches
T1L-H	7	T1 high-order latches
T2C-L	8	T2 low-order latches/counter
T2C-H	9	T2 high-order counter
SR	10	shift register
ACR	11	auxiliary control register
PCR	12	peripheral control register
VIFR	13	interrupt flag register
VIER	14	interrupt enable register
PTANH	15	as reg.no.1 except no handshake

2. Device address

<Device address> is specified as:

device address	meaning
HEX address	literal base address of device
<variable>%	integer containing device base address (in LS 16-bits)

and examples are:

```
&EEB0
```

and:

```
tempr%
```

where 'tempr%' would have previously been assigned to as in:

```
tempr% = &EEB0
```

3. Bit-pattern

<bit-pattern> may take any of the forms:

bit-pattern	meaning
bit-mask	individual bit settings e.g. %01101010
HEX value	bit settings expressed in HEX e.g. &6A (LS byte used)
<variable>%	integer containing bit-pattern (in LS byte)

so that the following would be equivalent:

```
%01110101

&76

bitmask%
```

where 'bitmask%' has previously been assigned to as in:

```
bitmask% = &76
```

Note that the bits to be operated on are always indicated by '1's, so that the bit-mask:

```
%00000100
```

would set the 2nd LS bit (LS bit = bit-0, MS bit = bit-7) when used with SET, but would unset the same bit when used with UNSET.

4. Location

<location> may be defined as:

location	meaning
HEX address	absolute address
<variable>%	integer to receive value (in LS bytes)

and examples are:

```
&80 (zero page location)

bitsfound%
```

When a variable is quoted, the obtained value is placed into the LS byte(s) of the variable, with other bytes zeroised. The result

is similar to an assignment statement such as:

```
bitsfound% = ?&EEB0
```

i.e. 'bitsfound%' will be assigned a value in the range &00 to &FF.

Summary of parameter types

A summary of the parameter types which may be quoted is shown in the table below. It is important to appreciate the interpretation of each type as described above. In the table, the following abreviations are used :

abbreviation	parameter type	maximum size
bits	bit-pattern e.g. %10011	8-bits i.e. 1 byte
HEX	HEX e.g. &FC03	2 bytes
var	<variable>% e.g. humid%	4 bytes

	parameter number			
procedure	1	2	3	4
CONFIGURE	PORT	HEX var	bits HEX var	
SETBITS	PORT	HEX var	bits HEX var	
UNSET	PORT	HEX var	bits HEX var	
TGGLE	PORT	HEX var	bits HEX var	
SHIFTL	PORT	HEX var		
SHIFTR	PORT	HEX var		
FETCH	PORT	HEX var	bits HEX	HEX var

Indirection operators

Word indirection operators are allowed, so that:

```
!addr%
```

is valid, although:

```
$addr%    (string indirection)
```

is not. Because the 'result' of the word indirection operator is 4 bytes i.e. an integer, it may be used with real values, as in:

```
!addr
```

and even:

```
!VAL(address$)
```

Note that in the case of FETCH in which a value is obtained, if an integer variable or word indirection operator is quoted to receive the value, the value obtained will be placed into the least significant byte, other bytes being cleared (i.e. zeroised).

Byte indirection operators (which access a single byte rather than a 4-byte integer) are allowed e.g. in parameters nos. 3 and 4 of the VIA handlers, but not in parameter no. 2 (where a 2-byte device base address is expected), i.e. they are allowed where a single byte value is required. The call

```
SETBITS[DDRB;!DEV1;&8A]
```

is allowed, whereas

```
SETBITS[DDRB;?DEV1;&8A]
```

is not.

Note that if a variable is quoted to receive a result using FETCH, if the variable does not exist, then it will be created.

Example procedure calls

We may wish to identify our physical device by a suitable name e.g:

```
humid% = &EEB0
```

could set the base address of our VIA handling a humidity controller. We may wish to initially 'configure' the device handler with:

```
CONFIGURE[DDRA;humid%;%10110110]
```

and subsequently manage it with calls such as:

```
SETBITS[ACR;humid%;%1000]
```

and:

```
TGGLE[PORTA;humid%;%11110]
```

Note that array elements may be used (always integers) as in:

```
FETCH[PORTA;humid%(2,temp%-32);masks%(temp%);rating%(temp%)]
```

(where we are using a selection of devices !). Note that in this example, the 2-dimensional array 'humid%' would contain base addresses for our humidity controllers, and that the 1-dimensional array 'masks%' will contain a selection of bit-masks; the 1-dimensional array 'rating%' is intended to receive data from PORTA of the devices.

Section 4. The System Timers

The System Timers

After switch-on or after the command:

*EVENTSON

five counter/timers in memory are incremented every 1 centi-second.
These have been given the titles of timers 1-5. Each is only 4 bytes in size, so as to be compatible with a BASIC integer variable, and two procedures are provided for Read/Write of these:

SET_TIMER	sets specified timer (value in centiseconds)
RD_TIMER	reads specified timer

The syntax required for correct operation is:

SET_TIMER[<timer no.>;<centisecs>]

RD_TIMER[<timer no.>;<centisecs>]

Parameters

The timer number must be specified as a literal DECIMAL number in the range 1-5 inclusive. Specifying the Read/Write of a timer outside this range will result in the message:

Bad Timer

being given.

Writing a Timer

When Writing to a timer, the value of the centi-seconds to write must be specified as the contents of an integer variable which has the required value assigned to it, as in:

```
time% = 100
```

and:

```
time% = -100
```

and the range is -2147483648 to +2147483647.

Reading a Timer

When reading the current value of a specified timer, an integer variable must be quoted to receive the result, if this variable does not already exist, then it will be created.

Time-out

Associated with all 5 timers is a system variable 'timflags' located at &64 in memory. The effect of a write to a timer is to clear an associated bit in this location, and conversely, when a timer 'times-out' i.e. passes through zero, the associated bit is set. All flags are set to zero immediately after the *EVENTSON command.

No function is provided in MULTI-BASIC for action based on the condition of this variable, so any required action must be defined by the user. For example it may be the basis for an 'event'.

The variable is bit-mapped as follows:

bit no.	Resulting condition after action
7,6,5	Reserved
4	0 : Write to Timer 5 1 : Timer 5 timed-out
3	0 : Write to Timer 4 1 : Timer 4 timed-out
2	0 : Write to Timer 3 1 : Timer 3 timed-out
1	0 : Write to Timer 2 1 : Timer 2 timed-out
0	0 : Write to Timer 1 1 : Timer 1 timed out

A typical 'event' using a timer might be:

```
120 TASK rdprt
130       FETCH[PORTA;&FE60;&FF;key%]
140       SET_TIMER[3;timeout%]
150 EXIT
160 REM set up timer 3 to cause an event
170 timeout% = -100 :REM 100 centi-seconds before time-out
180 SET_TIMER[3;timeout%]
190 evnt% = &00640400 :REM flag address = &64, test bit 2, no
inversion
200 ENABLE rdprt WHENEVER evnt%
```

It is important to note that the bit will remain set until a write to the relevent timer is made. Had we not reset the flag by re-writing timer 3 in the above example, the task would re-assert itself forever, and NOT at the correct time-interval. It is YOUR responsibilty to ensure that such conditions do not prevail !

Section 5. Command Summary

The Procedures and functions

Facilities provided by the additional procedures and functions are briefly described in the tables below.

VIA control

In each case the user specifies a device and register to be operated on.

procedure	action
CONFIGURE	configure the device/register by assigning a specified bit-pattern
SETBITS	set bits in device/register
UNSET	unset bits in a device/register
TGGLE	toggle bits in device/register
SHIFTL	shift device/register left
SHIFTR	shift device/register right
FETCH	get bits from device/register, place into specified location

System timer control

SET_TIMER	sets specified timer (value in centiseconds)
RD_TIMER	reads specified timer
WAIT	program suspended a specified number of ms (milliseconds)

Multi-tasking

Multi-tasking allows background tasks to be initiated both at intervals and on events. The foreground program is interrupted and suspended while the task(s) are running.
Tasks are declared in a similar way to procedures and functions, usually at the beginning of the user's program. They may subsequently be activated and de-activated, both by the foreground program and other tasks.

procedure	action
TASK	declares entry point for named task
EXIT	declares exit point of task
ENABLE	activates a named task at specified frequency or on an event
DISABLE	de-activates a task or all tasks
OMIT	drops a named task, or all tasks, from the current list

Functions are available to check on current tasks declared and enabled.

function	result
DECLD	integer value, indicating that specified task, or any task, is currently declared
ENABD	integer value, indicating that specified task, or any task, is currently enabled

Procedures, functions and their parameters

The syntax of the procedures and functions are as follows:

VIA handlers

CONFIGURE[<register>;<device-address>;<bit-pattern>]

SETBITS[<register>;<device-address>;<bit-pattern>]

UNSET[<register>;<device-address>;<bit-pattern>]

TGGLE[<register>;<device-address>;<bit-pattern>]

SHIFTL[<register>;<device-address>]

SHIFTR[<register>;<device-address>]

FETCH[<register>;<device-address>;
<bit-pattern>;<location>]

System timer handlers

SET_TIMER[<timer no.>;<centisecs>]

RD_TIMER[<timer no.>;<centisecs>]

WAIT[<millisecs>]

Multi-tasking

Free format commands:

TASK <task name>

EXIT

ENABLE <task name> EVERY <centisecs>cs

or

ENABLE (task name> WHENEVER <event identifier>

DISABLE <task name>

OMIT <task name>

Functions with parameter:

DECLD["<task name>"]

ENABD["<task name>"]

Parameter values

Parameters may be specified in the forms:

Register

Registers may be specified by one of the following names:

register	no.	R6522 designation
PORTB	0	ORB/IRB output/input register B
PORTA	1	ORA/IRA output/input register A
DDRB	2	DDRB data direction register B
DDRA	3	DDRA data direction register A
T1C-L	4	T1 low-order latches/counter
T1C-H	5	T1 high-order counter
T1L-L	6	T1 low-order latches
T1L-H	7	T1 high-order latches
T2C-L	8	T2 low-order latches/counter
T2C-H	9	T2 high-order counter
SR	10	shift register
ACR	11	auxiliary control register
PCR	12	peripheral control register
VIFR	13	interrupt flag register
VIER	14	interrupt enable register
PTANH	15	as reg.no.1 except no handshake

Device address

Device address is specified as:

device address	meaning
HEX address	literal base address of device
<variable>%	integer containing device base address (in LS 16-bits)

Bit-pattern

The bit-pattern may take any of the forms:

bit-pattern	meaning
bit-mask	individual bit settings e.g. %01101010
HEX value	bit settings expressed in HEX e.g. &6A (LS byte used)
<variable>%	integer containing bit-pattern (in LS byte)

Location

Location may be defined as:

location	meaning
HEX address	absolute address
<variable>%	integer to receive value (in LS bytes)

When a variable is quoted, the obtained value is placed into the LS byte(s) of the variable, with other bytes zeroised. The result is similar to an assignment statement such as:

```
bitsfound% = ?&EEB0
```

i.e. 'bitsfound%' will be assigned a value in the range &00 to &FF.

Pointer

Pointer may be defined as:

pointer	meaning
HEX address	literal address
<variable>%	integer which contains address to be used (in LS 2-bytes)

Length

Length may be quoted as:

length	meaning
HEX value	literal length
<variable>%	integer which contains length (in LS 2-bytes)

Millisecs

value	meaning
HEX value	literal time in ms
<variable>%	contains time in ms (in LS 2 bytes)

Centisecs

value	meaning

DEC value	literal time interval

Event identifier

value	meaning
<variable>%	MS 2 bytes : an address next LS byte : a bit pattern LS byte : invert pattern

Timer number

value	meaning
DEC	timer no. in decimal - 1 to 5 inclusive only allowed

Task name

value	meaning
chars	character string of up to 7 chars in length e.g. 'chktemp'
*	any task n.b. used with DECLD and ENABD within quotes i.e. as "*"

Parameters within square brackets

	parameter number			
procedure	1	2	3	4
CONFIGURE	PORT	HEX var	bits HEX var	
SETBITS	PORT	HEX var	bits HEX var	
UNSET	PORT	HEX var	bits HEX var	
TGGLE	PORT	HEX var	bits HEX var	
SHIFTL	PORT	HEX var		
SHIFTR	PORT	HEX var		
FETCH	PORT	HEX var	bits HEX	HEX var
WAIT	HEX var			
DECLD	"chars"			
ENABD	"chars"			

Free format parameters

These include the literal char strings 'EVERY', 'WHENEVER' and 'cs'.

	parameter number		
procedure	1	2	3
TASK	chars		
EXIT			
ENABLE	chars	'EVERY'	DEC'cs'
-or-			
ENABLE	chars	'WHENEVER	var%
DISABLE	chars		
OMIT	chars		

n.b. DISABLE, OMIT, DECLD and ENABD may include '*' as a task name - this indicates all tasks

Variable parameters

Variable parameters must be of integer type, so that names of variables used directly will always end with '%' (except with byte or word indirection - see below). Valid examples are

```
addr%
nmbr%(md1,md2)
```

while invalid names would be:

```
addr      (real type)
```

```
nmbr$    (string type)
```

Indirection operators

Word indirection operators are allowed, so that:

```
!addr%
```

is valid, although:

```
$addr%   (string indirection)
```

is not. Because the 'result' of the word indirection operator is 4 bytes i.e. an integer, it may be used with real values, as in:

```
!addr
```

and even:

```
!VAL(address$)
```

Note that in all cases in which a value is obtained (e.g. FETCH, RS_INT, etc) if an integer variable or word indirection operator is quoted to receive the value, the value obtained will be placed into the least significant byte(s), other bytes being cleared (i.e. zeroised).

Byte indirection operators (which access a single byte rather than a 4-byte integer) are allowed e.g. in parameters nos. 3 and 4 of VIA handlers but not in parameter no. 2 (where a 2-byte device base address is expected) : i.e. they are allowed where a single byte value is required. The call:

```
SETBITS[DDRB;!DEV1;&8A]
```

is allowed, whereas:

SETBITS[DDRB;?DEV1;&8A]

is not.

Variables to receive result

If a variable is quoted to receive a result from a procedure i.e. with:

procedure	parameter number
FETCH	4
RD_TIMER	2

if the variable does not exist, then it will be created.

Section 6. Errors & Exceptions

Section 6. Errors & Exceptions

The following is a table of Error messages generated following an error condition in MULTI-BASIC. Where it may not be clear as to what actually caused the error condition, a suggestion as to what to examine is given.

Error Number	Message given	Suggestion
&C0	Missing [	
&C1	Missing]	
&C2	Too few/many parameters	Check number of parameters reqd for keyword.
&C3	Bad parameter(s)	Check allowable parameter types, or for non-existent variable.
&C4	Statement syntax	
&C5	Missing spaces	
&C6	Bad TASK name	Task name too long or contains invalid character.
&C7	Missing EVERY/WHENEVER	
&C8	Bad TASK interval	Interval should be a decimal numeric string followed immediately with 'cs'.
&C9	Bad TASK event	Check for non-existent or invalid variable type.
&CA	TASK not alone	The declaration <TASK name> should be a statement on its own.
&CB	? not allowed	Byte indirection has been attempted where it is not valid to do so.
&CC	Too many TASKs	Only 8 'current' tasks are allowed.

&CD	No TASK	EXIT has been executed whilst not in a background task.
&CE	Can't find EXIT	A task has been defined with no matching EXIT.
&CF	No such TASK	A task name has been quoted that is not in the current list.

Other messages may be generated by BASIC itself e.g:

Mistake

or:

Array

etc., if you have used array elements or expression subscripts.

Note that 'Array' is often generated when round brackets are used incorrectly.

You should check the following details of your program statement:

For the command line :
- command spelt correctly
- there are no spaces within the command name and any parameters enclosed in brackets
- square brackets have been used
- parameters are separated by ';'

Parameters are compatible with the command :
- correct number of parameters have been quoted

For each parameter quoted :
- register identity is spelt correctly
- HEX addresses are in range >= &0, <= &FFFF, and has no more

than 4 HEX digits
- bit-patterns start with '%', use only 0s and 1s, and have no more than 8 binary digits
- variables : - are integers (end with '%')
  - conform to the BBC's naming convention
  - names (including any array element subscripts) are <= 65 characters in length
  - array elements are correctly subscripted

With TASK, ENABLE, DISABLE & OMIT :
- spaces separate the command and task name etc.
- task name has between 1 - 7 characters
- time intervals are valid
- event status variable name conforms to usual BASIC, and is <= 65 characters in length
- event status variable actually exists

Section 7. DOs, DON'Ts & AVOIDs

DO remember that if you require multi-tasking, to enable interrupts with *EVENTSON, if these have been previously disabled with *EVENTSOFF.

DO keep tasks to a reasonable length.

AVOID using REM statements inside a TASK .... EXIT construct - program speed will be affected. If you wish to document the task, do this OUTSIDE the task body.

DON'T use real variables if you can avoid them - integer calculations are much faster.

DO use short variable names as an aid to faster program execution, and remember their maximum length in a MULTI-BASIC statement is 65 characters.

AVOID using PROC calls from within tasks - program performance will suffer.

DO remember it is YOUR responsibility to clear any status causing an 'event' (usually by reading or writing a relevant register).

DON'T use WAIT unless you really mean it - ALL program activity is suspended.

DON'T use the area &50-&8F - this is used by MULTI-BASIC.

DON'T re-vector the BRK vectors unless you can return control to the previous owner correctly.

DO remember to use *EVENTSOFF and *EVENTSON when CHAINing programs

Section 8. Further Examples

Data logging

This example illustrates data collection from 4 ADC channels, with graphic display, and subsequent data logging onto a disc file for future analysis.

```
 10 REM ****************************
 20 REM     example of data logging
 30 REM ****************************
 40 REM
 50 REM arrays to hold data and plotting coordinates
 60 REM
 70 DIM data%(60,4)
 80 DIM coord%(4,2)
 90 REM
100 REM declare task
110 REM
120 TASK data
130    REM trace the task
140    x1%=r% MOD 8:r%=r%+1:x2%=r% MOD 8
150    PRINT TAB(10+x2%,0);"*";
160    PRINT TAB(10+x1%,0);" ";
170    VDU 7
180    REM read from ADC channels and plot on screen
190    sec% = (r% MOD 60) + 1
200    FOR chan% = 1 TO 4
210       data%(sec%,chan%) = ADVAL(chan%)
220 data%(sec%,chan%) = ((chan%-1) * 16*1024) + (sec% * 200)
230       GCOL 0,chan%
240       MOVE coord%(chan%,1),coord%(chan%,2)
250       DRAW sec%*20,data%(sec%,chan%)/80
260       coord%(chan%,1) = sec%*20
270       coord%(chan%,2) = data%(sec%,chan%)/80
280    NEXT chan%
290    REM log data at intervals
300    IF sec%=60 THEN PROClogging
```

```
310 EXIT
320 REM
330 REM display appropriate screen
340 REM
350 MODE 2
360 VDU 24,0;0;1279;800;
370 MOVE 0,0
380 PRINT TAB(0,0);"data     :";
390 PRINT TAB(0,1);"logging :";
400 REM
410 REM preset plotting coordinates
420 REM
430 FOR chan% = 1 TO 4
440     coord%(chan%,1) = 0
450     coord%(chan%,2) = 0
460 NEXT chan%
470 REM
480 REM open data log file for output
490 REM
500 file% = OPENOUT("datafle")
510 r%=0:g%=0
520 ENABLE data     EVERY  100cs
530 REM
540 REM loop until key is hit
550 REM
560 REPEAT
570     key% = INKEY(10)
580 UNTIL key%<>-1
590 CLOSE#file%
600 STOP
610 REM
620 REM logging procedure
630 REM
640 DEF PROClogging
650     REM trace logging
660     x1%=g% MOD 8:g%=g%+1:x2%=g% MOD 8
670     PRINT TAB(10+x2%,1);"*";
680     PRINT TAB(10+x1%,1);" ";
```

```
690     REM write to disc file
700     FOR sec% = 1 TO 60
710        FOR chan% = 1 TO 4
720           PRINT#file%,data%(sec%,chan%)
730        NEXT chan%
740     NEXT sec%
750     REM clear graphics (plot) area
760     CLG
770     REM reset X-coordinate
780     FOR chan% = 1 TO 4
790        coord%(chan%,1) = 0
800     NEXT chan%
810 ENDPROC
```

The ADC channels return a value in the range 0 to 65520. This is 'simulated' in line 220 - if you've some games paddles, plug them in and remove line 220.

Enhancements to this program could include the display of the Y coordinate scale, and X coordinate timescale.

Tracing

In addition to the TRACE facility available in ordinary BASIC, the user may wish to have a separate indication of when each task is executed. The following example program shows a simple means of providing display indicators for this purpose.

```
 10 REM ****************************
 20 REM     example of task tracing
 30 REM ****************************
 40 TASK red
 50    x1%=r% MOD 16:r%=r%+1:x2%=r% MOD 16
 60    PRINT TAB(20+x2%,0);"*";
 70    PRINT TAB(20+x1%,0);" ";
 80 EXIT
 90 TASK green
100    x1%=g% MOD 16:g%=g%+1:x2%=g% MOD 16
110    PRINT TAB(20+x2%,1);"*";
120    PRINT TAB(20+x1%,1);" ";
130 EXIT
140 TASK yellow
150    x1%=y% MOD 16:y%=y%+1:x2%=y% MOD 16
160    PRINT TAB(20+x2%,2);"*";
170    PRINT TAB(20+x1%,2);" ";
180 EXIT
190 TASK blue
200    x1%=b% MOD 16:b%=b%+1:x2%=b% MOD 16
210    PRINT TAB(20+x2%,3);"*";
220    PRINT TAB(20+x1%,3);" ";
230 EXIT
240 ENABLE red    EVERY  25cs
250 ENABLE green  EVERY  50cs
260 ENABLE yellow EVERY 100cs
270 ENABLE blue   EVERY 200cs
280 MODE 7
290 r%=0:g%=0:y%=0:b%=0
```

```
300 PRINT TAB(10,0);CHR$(129);"red     :";
310 PRINT TAB(10,1);CHR$(130);"green   :";
320 PRINT TAB(10,2);CHR$(131);"yellow  :";
330 PRINT TAB(10,3);CHR$(132);"blue    :";
340 PRINT TAB(5,10);CHR$(134);"Example of TASK tracing"
350 PRINT TAB(5,11);CHR$(134);"======================"
360 PRINT TAB(3,13);CHR$(134);"A moving asterisk indicates"
370 PRINT TAB(3,14);CHR$(134);"a currently ENABLEd TASK"
380 REPEAT
390 UNTIL 0
400 STOP
```

Menu control of tasks

The following example illustrates one way of task control from a screen menu.

```
  10 REM ****************************
  20 REM     example of task tracing
  30 REM ****************************
  40 REM
  50 REM declare 4 tasks
  60 REM
  70 TASK flash
  80    x1%=r% MOD 16:r%=r%+1:x2%=r% MOD 16
  90    PRINT TAB(20+x2%,0);"*";
 100    PRINT TAB(20+x1%,0);" ";
 110    IF r% MOD 2 THEN 150
 120    PRINT TAB(20,7);CHR$(136);CHR$(141);"f l a s h";
 130    PRINT TAB(20,8);CHR$(136);CHR$(141);"f l a s h";
 140    GOTO 170
 150    PRINT TAB(20,7);"            ";
 160    PRINT TAB(20,8);"            ";
 170 EXIT
 180 TASK colours
 190    x1%=g% MOD 16:g%=g%+1:x2%=g% MOD 16
 200    PRINT TAB(20+x2%,1);"*";
 210    PRINT TAB(20+x1%,1);" ";
 220    PRINT TAB(20,10);" c o l o u r s";
 230    PRINT TAB(20,11);" c o l o u r s";
 240    FOR col%=0 TO 6
 250       PRINT TAB((20+(col%*2)),10);CHR$(129+((g%+col%) MOD
7));
 260       PRINT TAB((20+(col%*2)),11);CHR$(129+((g%+col%) MOD
7));
 270    NEXT col%
```

```
  280 EXIT
  290 TASK beep
  300    x1%=y% MOD 16:y%=y%+1:x2%=y% MOD 16
  310    PRINT TAB(20+x2%,2);"*";
  320    PRINT TAB(20+x1%,2);" ";
  330    VDU 7
  340 EXIT
  350 TASK alarm
  360    x1%=b% MOD 16:b%=b%+1:x2%=b% MOD 16
  370    PRINT TAB(20+x2%,3);"*";
  380    PRINT TAB(20+x1%,3);" ";
  390    ENVELOPE 2,1,2,-2,2,10,20,10,1,0,0,-1,100,100
  400    SOUND 1,2,100,100
  410 EXIT
  420 REM
  430 REM display appropriate screen
  440 REM
  450 MODE 7
  460 PRINT TAB(10,0);CHR$(129);"flash    :";
  470 PRINT TAB(0,7);CHR$(129);"flash    :";
  480 PRINT TAB(0,8);CHR$(129);
  490 PRINT TAB(10,1);CHR$(130);"colours :";
  500 PRINT TAB(0,10);CHR$(130);"colours :";
  510 PRINT TAB(0,11);CHR$(130);
  520 PRINT TAB(10,2);CHR$(131);"beep     :";
  530 PRINT TAB(0,13);CHR$(131);"beep     :";
  540 PRINT TAB(0,14);CHR$(131);
  550 PRINT TAB(10,3);CHR$(132);"alarm    :";
  560 PRINT TAB(0,16);CHR$(132);"alarm    :";
  570 PRINT TAB(0,17);CHR$(132);
  580 PRINT
TAB(0,4);CHR$(134);"==================================";
  590 PRINT TAB(1,5);CHR$(134);"TASK     STATUS          ACTION";
  600 PRINT TAB(1,6);CHR$(134);"____     ______          ______";
  610 PRINT
TAB(1,20);CHR$(134);"==================================";
  620 PRINT TAB(1,21);CHR$(134);"* select a TASK with up/down
arrow";
```

```
  630 PRINT TAB(1,22);CHR$(134);"* enter : E (ENABLE) + 1-9
(seconds)";
  640 PRINT TAB(1,23);CHR$(134);"       or : D (DISABLE)";
  650 PRINT
TAB(1,24);CHR$(134);"==================================";
  660 FOR T% = 1 TO 4
  670     PRINT TAB(11,(7+((T%-1)*3)));"E
";CHR$((2^(T%-1))+&30);CHR$(137);":";
  680 NEXT T%
  690 T% = 1
  700 PRINT TAB(10,(7+((T%-1)*3)));CHR$(136);
  710 REM
  720 REM preset indicators and enable tasks with initial
intervals
  730 REM
  740 r%=0:g%=0:y%=0:b%=0
  750 ENABLE flash   EVERY 100cs
  760 ENABLE colours EVERY 200cs
  770 ENABLE beep    EVERY 400cs
  780 ENABLE alarm   EVERY 800cs
  790 REM
  800 REM enable use of cursor keys
  810 REM
  820 *FX 4,1
  830 REM
  840 REM now accept input to control tasks
  850 REM n.b. RETURN stops program
  860 REM
  870 REPEAT
  880     REM get control character
  890     C% = 0
  900     C% = INKEY(100)
  910     REM branch accordingly
  920     IF C%=&8A THEN 970
  930     IF C%=&8B THEN 1020
  940     IF C%=&44 THEN 1070
  950     IF C%=&45 THEN 1190
  960     GOTO 1640
```

```
 970     REM step to next task
 980     PRINT TAB(10,(7+((T%-1)*3)));" ";
 990     T% = T%+1 : IF T%=5 THEN T%=1
1000     PRINT TAB(10,(7+((T%-1)*3)));CHR$(136);
1010     GOTO 890
1020     REM step back to previous task
1030     PRINT TAB(10,(7+((T%-1)*3)));" ";
1040     T% = T%-1 : IF T%=0 THEN T%=4
1050     PRINT TAB(10,(7+((T%-1)*3)));CHR$(136);
1060     GOTO 890
1070     REM disable selected task
1080     PRINT TAB(11,(7+((T%-1)*3)));"D";
1090     PRINT TAB(14,(7+((T%-1)*3)));" ";
1100     ON T% GOTO 1110,1130,1150,1170
1110     DISABLE flash
1120     GOTO 890
1130     DISABLE colours
1140     GOTO 890
1150     DISABLE beep
1160     GOTO 890
1170     DISABLE alarm
1180     GOTO 890
1190     REM enable selected task
1200     I% = INKEY(100)
1210     IF I%<&31 OR I%>&39 THEN 1200
1220     PRINT TAB(11,(7+((T%-1)*3)));"E";
1230     PRINT TAB(14,(7+((T%-1)*3)));CHR$(I%);
1240     E% = (T%-1)*9 + (I%-&31) + 1
1250     ON E% GOSUB
1280,1290,1300,1310,1320,1330,1340,1350,1360,1370,1380,1390,1400,
1410,1420,1430,1440,1450,1460,1470,1480,1490,1500,1510,1520,1530,
1540,1550,1560,1570,1580,1590,1600,1610,1620,1630
1260     GOTO 890
1270     REM table of enables
1280     ENABLE flash   EVERY 100cs : RETURN
1290     ENABLE flash   EVERY 200cs : RETURN
1300     ENABLE flash   EVERY 300cs : RETURN
1310     ENABLE flash   EVERY 400cs : RETURN
```

```
1320     ENABLE flash   EVERY 500cs : RETURN
1330     ENABLE flash   EVERY 600cs : RETURN
1340     ENABLE flash   EVERY 700cs : RETURN
1350     ENABLE flash   EVERY 800cs : RETURN
1360     ENABLE flash   EVERY 900cs : RETURN
1370     ENABLE colours EVERY 100cs : RETURN
1380     ENABLE colours EVERY 200cs : RETURN
1390     ENABLE colours EVERY 300cs : RETURN
1400     ENABLE colours EVERY 400cs : RETURN
1410     ENABLE colours EVERY 500cs : RETURN
1420     ENABLE colours EVERY 600cs : RETURN
1430     ENABLE colours EVERY 700cs : RETURN
1440     ENABLE colours EVERY 800cs : RETURN
1450     ENABLE colours EVERY 900cs : RETURN
1460     ENABLE beep    EVERY 100cs : RETURN
1470     ENABLE beep    EVERY 200cs : RETURN
1480     ENABLE beep    EVERY 300cs : RETURN
1490     ENABLE beep    EVERY 400cs : RETURN
1500     ENABLE beep    EVERY 500cs : RETURN
1510     ENABLE beep    EVERY 600cs : RETURN
1520     ENABLE beep    EVERY 700cs : RETURN
1530     ENABLE beep    EVERY 800cs : RETURN
1540     ENABLE beep    EVERY 900cs : RETURN
1550     ENABLE alarm   EVERY 100cs : RETURN
1560     ENABLE alarm   EVERY 200cs : RETURN
1570     ENABLE alarm   EVERY 300cs : RETURN
1580     ENABLE alarm   EVERY 400cs : RETURN
1590     ENABLE alarm   EVERY 500cs : RETURN
1600     ENABLE alarm   EVERY 600cs : RETURN
1610     ENABLE alarm   EVERY 700cs : RETURN
1620     ENABLE alarm   EVERY 800cs : RETURN
1630     ENABLE alarm   EVERY 900cs : RETURN
1640 UNTIL C%=&0D
1650 STOP
```

Section 9. System use

User VIA

MULTI-BASIC derives its control from Timer 2 of the User VIA (mapped at &FE60 in the BBC Machine). The user is warned not to write to this timer, or other VIA registers which may affect the operation of the timer interrupts, without being fully aware of the consequences of so doing.

Zero-page use

MULTI-BASIC uses zero page workspace from &50 to &8F inclusive. If you require to use zero-page from within a BASIC program, i.e. from a short section of machine-code you are advised only to use those addresses marked with R/W. The contents MUST be saved on the stack at the entry point of your code, and restored on exit. Those locations marked with an asterisk SHOULD NOT BE VIOLATED AT ALL, otherwise the system may crash, or at best give spurious results. You should also disable interrupts on entry to your code, if you require correct operation of MULTI-BASIC. You must, of course, re-enable these on exit.
Don't forget that if you are not using ECONET, locations &90-&9F are free, and also &A0-&A7, when Disc access is not required. Where appropriate the address is given with an indication of its use.

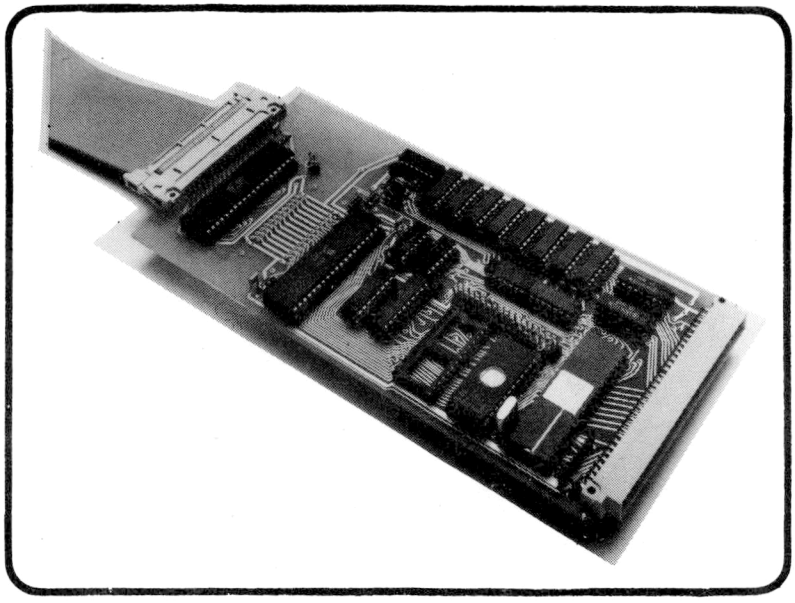
&50-&51	*
&52	Decremented every 1cs. R/W
&53	*
&54	Tasks to be executed in current time slot. Read only
&55-&5F	*
&60-&63	R/W
&64	Timer 'time-out' flags. R/W
&65	*
&66	Tasks overrun. Read only
&67-&6F	*
&70-&8F	R/W

System workspace

MULTI-BASIC also uses 3 pages of private workspace in main RAM. This, however is dynamically allocated by the Operating system, so it is impossible to predict where this is in advance. Locations &5B-&5C contain a pointer to the start of this area, and you are advised not to violate it.

Single Board 6809 2nd Processor for the BBC Micro

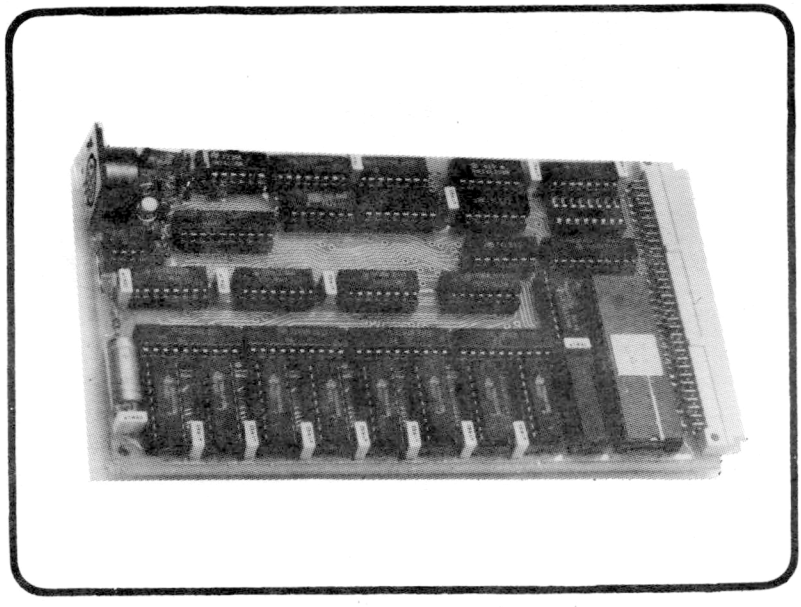
* Enables standard FLEX format discs to run on BBC.
* Supports high level language compilers.
* Cross assemblers and disassemblers for most micros.
* Connects directly onto the tube.
* Sits inside the BBC or plugs into an extension rack.
* 64K Dynamic ram on board.
* Two 28 Pin Byte wide memory sockets.
* Optional battery back up for CMOS devices.



CMS Ltd. Tel. 0223-324141

High Performance Colour Graphics Card

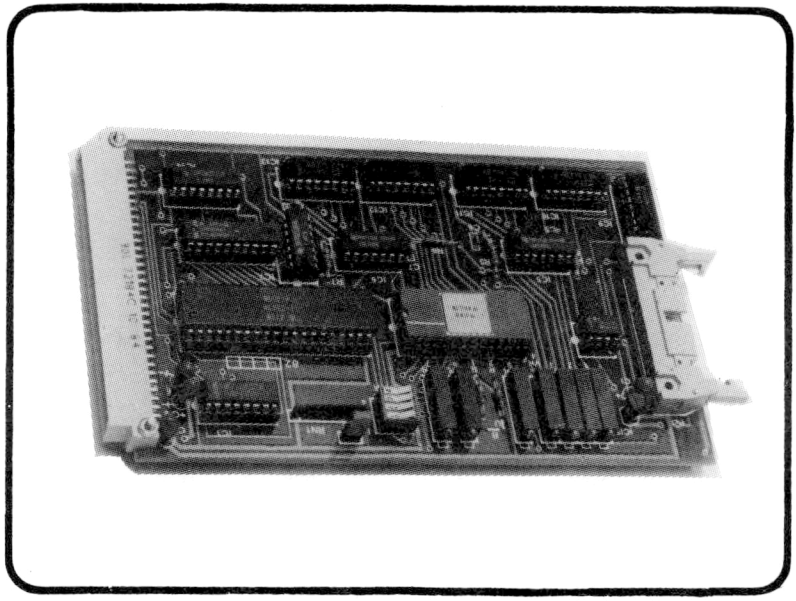
* 64K of on-board graphics memory.
* Universal RGB TTL levels.
* Composite video output.
* High resolution 512x256 pixels.
* 4 colour planes.
* Simultaneous black & white colour display.
* Easy software implementation.
* X and Y screen addressing by 12 Bit registers.
* Hardware ASCII character generator.
* Unique zoom facility.
* Internal light pen facility.
* Hardware vector generator capable of drawing 1.5 million dots per second.
* Full address decoding.



CMS Ltd. Tel. 0223-324141

12 Bit Analogue Interface Card

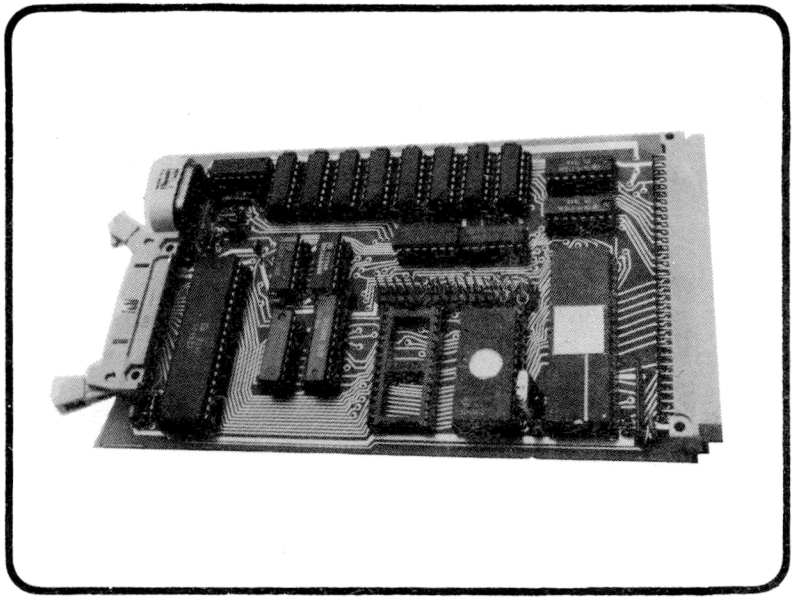
* True 12 Bit resolution.
* 8 Multiplexed 12 bit input channels.
* 25us typical conversion speed.
* 7 uncommitted digital I/O channels.
* Unipolar - Bipolar operation upto 10v input.
* Temperature range 0-50C.
* Fully buffered 8 Bit data bus.
* Relocatable 6809 software available.



CMS Ltd. Tel. 0223-324141

6809 Controller for Industrial Applications

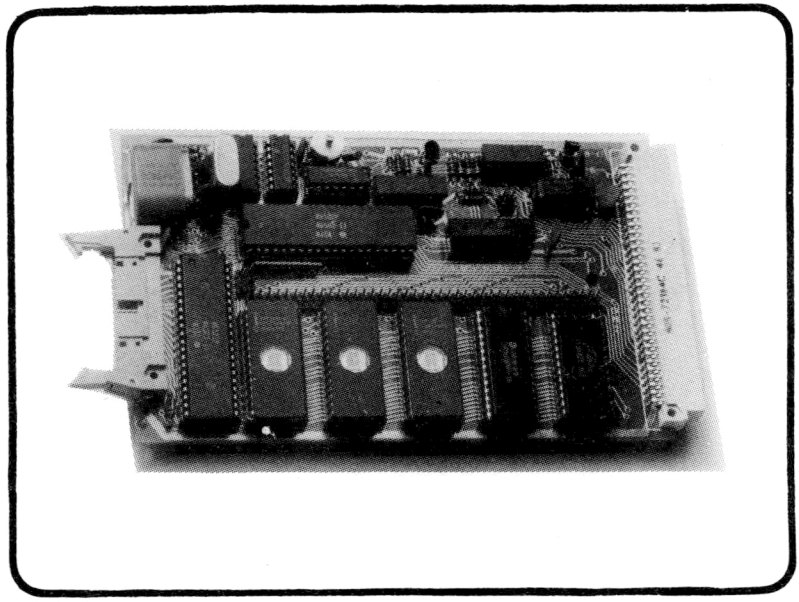
* 6809 advanced 8 bit microprocessor.
* 20 I/O lines.
* Two timers.
* 64K dram onboard.
* Supports up to 32K ROM.
* Battery backup for CMOS RAM.
* Dual software switched memory map.
* Programmable address decoding.
* Machine operating system included in price.
* Supported by a full range of software packages.



CMS Ltd. Tel. 0223-324141

Universal Controller with a choice of Processors - 6502/6809

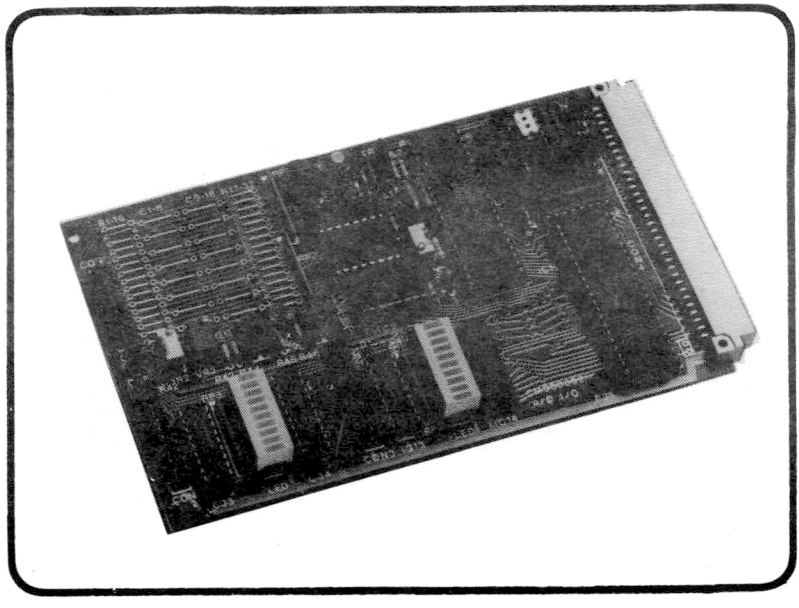
* Choice of processor 6502 or 6809.
* Dynamic memory management.
* Battery backup CMOS RAM.
* Real time clock.
* Supports 64K RAM or PROM.
* Easy to use.
* 20 I/O lines.
* 5 Byte wide memory sockets.



CMS Ltd. Tel. 0223-324141

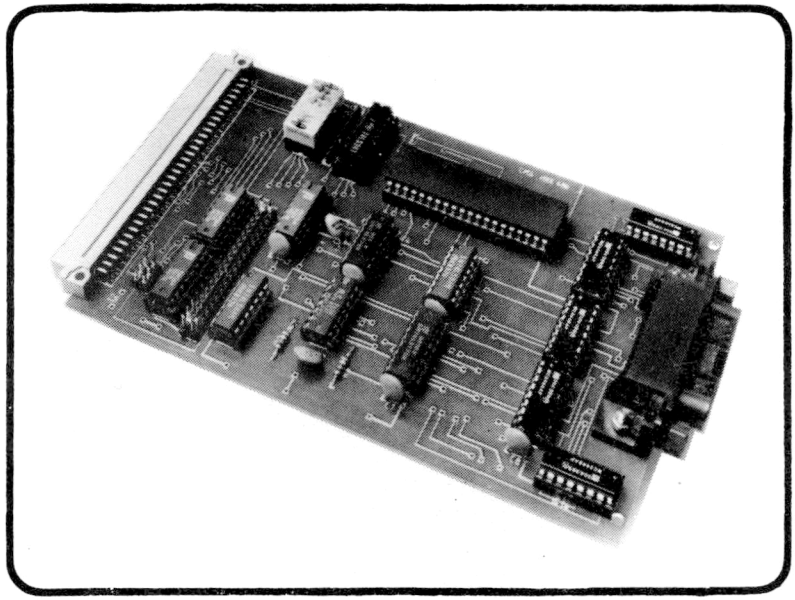
Analogue/Digital Interface

* 16 High performance 13 Bit ADC input channels.
* Actually stable to 13 Bits.
* 12 Bit DAC with potentiometer controlled offset & gain.
* Capable of Unipolar-Bipolar operation.
* 16 digital output lines can drive 50v each at .5 amp.
* All output lines have LED indicators. * 4 digital input lines upto 100v.
* 2 user programmable timer counters.
* Ideal for control and teaching.
* Free MULTI-BASIC software drivers.
* Free PL9 software drivers.



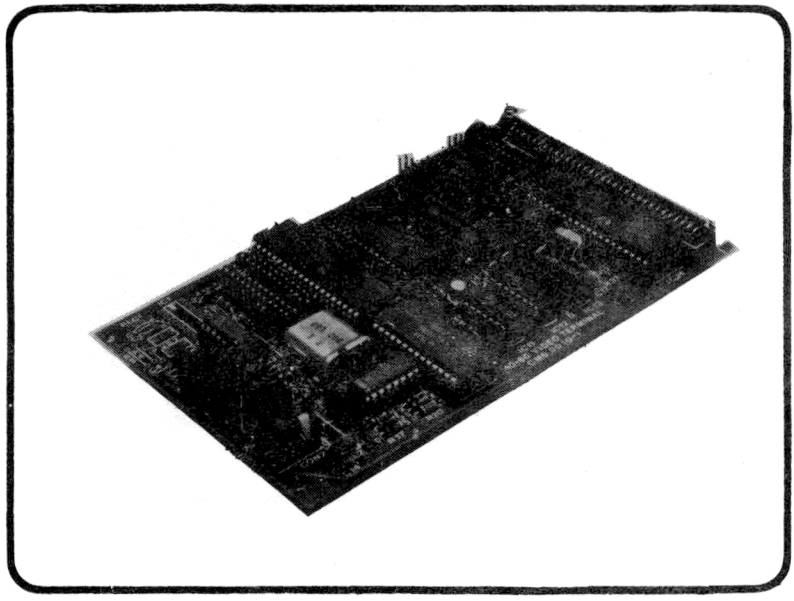
IEEE Talker/Listener Controller

* Full talker/listener controller capability.
* Conforms to IEEE 488 1978 specification.
* Suitable for communication with upto 14 devices.
* Complete software drivers written in PL9.
* Serial/parallel poll.
* Talk only / listen only capability.
* Single or dual primary address recognition.
* Programmable interrupts.



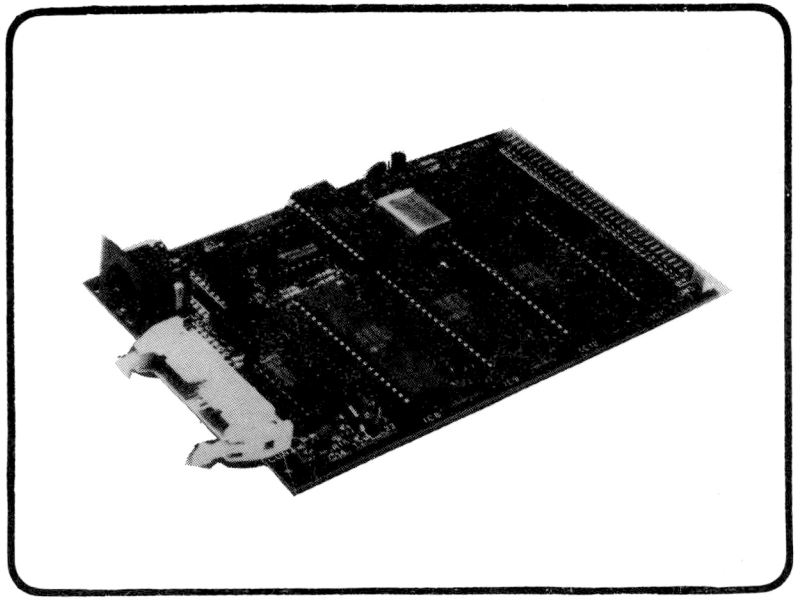
40/80 Video Terminal

* Full colour 40 or 80 column.
* Exhanced teletext character set.
* Hardware sroll capability.
* Serial interface RS423/422.
* Video control occupies only 16K of system memory.
* 8K video memory expandable to 16K.
* Genlock option available.
* Attributes : underlining, flashing, reverse video.
* User definable characters.
* Pixel graphics.
* VIA interface.
* Centronics printer port.
* Free MULTI-BASIC software drivers.
* Free software drivers written in PL9.



Versatile Interface Board

* 4 versatile interface adaptors providing 80 independent digital input/ouput control lines.
* Easy interfacing to 26 way IDC connector.
* 8-16 bit programmable timer counters
* ACIA providing serial communications at rates between 50 to 19,200 baud
* RS422-RS423 interface buffering for high speed communications in noisy environments.
* Programmable independent transmit and receive clock capability.
* Serial output buffers may be programmed to go tristate for distributed multi-process applications.
* True centronics interface.
* Full on-board address decoding.
* Available in a comprehensive range of options when purchased in quantity.



MULTI-BASIC

Already used in demanding industrial environments, MULTI-BASIC is a well proven system providing BBC BASIC users with the ability to write and run Multi-tasking programs on the BBC Micro-computer.

The multi-tasking facilities have been designed with ease of use and clarity of purpose in mind, and in no way interfere with the operation of 'normal' BASIC functions, but serve to convert BBC BASIC into what is undeniably the most powerful BASIC interpreter in use on small machines.

For those wishing to complement or extend the physical abilities of the BBC Microcomputer, Cambridge Microprocessor Systems offer both a 1 Mhz Bus expansion board, and also a 6502 2nd processor, together with a wide range of Eurocards, all of which are supported by a full implementation of MULTI-BASIC.

Cambridge Microprocessor Systems Software Ltd.,
Unit 18,
Industrial Estate,
Chelmsford Road,
GREAT DUNMOW,
ESSEX CM6 1XG