

Micropak 870C

**An advanced simulator based development tool for
Toshiba TLCS-870/C microcontrollers**

User manual

Document reference: Micropak 870C User manual v1.0 July 2001

Copyright

Copyright 2001 AND Software Ltd., All rights reserved. No part of this publication may be reproduced, transmitted, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, optical, chemical, manual or otherwise, without the prior written permission of AND Software Ltd., 4 Forest Drive, Theydon Bois, Epping, Essex CM16 7EY, England.

Disclaimer

AND Software Ltd. makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, AND Software Ltd. reserves the right to make changes from time to time in the content hereof without obligation of AND Software Ltd. to notify any person of such revision or changes.

Accuracy

AND Software Ltd. cannot guarantee the accuracy or functionality of the simulator under any particular circumstance. Neither AND Software Ltd. nor any of its agents shall be liable in any way for any losses incurred wholly or partly as a consequence of any errors, omissions or assumptions made in or by the simulator or its associated documentation.

AND SOFTWARE LTD

4 Forest Drive
Theydon Bois
Essex, CM16 7EY
England

Tel: +44 (0)1992 814655 • Fax: +44 (0)1992 813362
Email: AND@andsoftware.co.uk
Homepage: www.andsoftware.co.uk

Contents

About this guide	ix
INTRODUCTION	1
The TLCS-870/C microcontroller family	1
Micropak 870C - An overview	1
TLCS-870/C devices supported by Micropak 870C	3
INSTALLATION	5
The important files	5
Suggested directory structure	5
Running the SETUP utility	6
Run time links to the 870/C tool chain	7
FAST STARTUP GUIDE	9
Starting the Micropak 870C simulator	9
Setting up and debugging a new project	9
Opening an existing project	10
Common debugging features explained	10
TUTORIAL	15
The example application - stepper motor control	15
Setting up the test environment	16
Understanding the example	27
Executing the code - The PC indicator and cursor	31
Simulating button presses	34
Project context files	34
Using the script file	36
Checking the generated sequence	36
Tracking down the bug	36
Making corrections	39
Moving on	39
USER INTERFACE DETAILS	41
The Micropak 870C screen	41
Window elements	43
Menu operation	45
Menu function reference descriptions	46
Using dialog boxes	51

The tool bar	53
Using the on-line help system	54
NAVIGATING PROJECT FILES	57
Project file overview	57
Opening a project	59
Files Window	60
Editing a project	60
Specifying project settings	61
USING THE EDITOR	67
Opening files	67
Syntax colouring	68
Mouse driven functions when editing	69
Editor options	70
Keyboard functions when editing	70
Locating and changing text	71
The implications of editing	74
Re-building the project	74
CONTROLLING EXECUTION	77
Overview	77
Execution possibilities	77
Resetting and viewing the processor clock	79
The program counter	80
The interval window	80
Debug options	81
TRACE OPTIONS	83
Trace Buffering	83
Controlling Tracing	83
Trace buffer displays	84
Restarting execution	85
Inactive trace buffer	85
Performance Analysis	85
Code Coverage	86
BREAKPOINTS	87
Setting breakpoints	87
The 'Type' field	88
The 'Location' field	89
The 'Expression/L-Value' field	89

The 'Length' field	90
The 'Counter' field	90
Adding a breakpoint	90
Viewing current breakpoints set ups	90
Setting breakpoints in the source window	91
Removing breakpoints	91
Enabling/disabling breakpoints	91
Script file facilities	92
PORT SIMULATION TECHNIQUES	93
Overview	93
Using script files to control port conditions	95
Using script files to check port conditions	96
Pin numbering	96
VIEWING SIMULATED OBJECTS	97
Overview	97
Displaying RAM	97
Signal recording boxes	98
On-chip peripherals	102
Port views	104
Test panel displays	104
SOURCE DEBUGGING	111
Overview	111
Source Windows	111
Quick Watch	112
Watch	113
Locals	114
Call Stack	114
Registers	115
Browser	115
USING SCRIPT FILES	117
Overview	117
Script files - Purpose and uses	117
The script language	119
Script file variables	120
Script operators and expressions	121
Script file execution and control flow	123
Identities in script files	127
Script keywords	128

Script file commands and functions - detailed descriptions	131
ABS	131
ACOS	131
AND	132
ASC	132
ASIN	132
ATAN	133
BREAKPOINT	133
CHR\$	135
CLOSE	135
CONNECT	135
COS	136
DIM	136
EDIT	136
END	137
EQV	137
EVENT	137
EXP	139
FOR - TO - [STEP]	139
GETEDIT(\$)	140
GETFC	140
GETFS	140
GETrr	141
GETTIME	141
GETV	141
GLOBAL	142
GO	142
GOSUB	142
GOTO	143
IF - THEN - [ELSEIF] - [ELSE] - ENDIF	143
IMP	144
INPUT	144
LEFT\$	144
LEN	145
LET	145
LOCAL	146
MID\$	146
NOT	147
OPEN	147
OR	148

PEEK	148
PRINT	148
POKE	149
REM	149
REPEAT - UNTIL	150
RETURN	150
RIGHT\$	150
SETBITMAP	151
SETEDIT	151
SETFLAG	152
SETrr	152
SETR	152
SETV	153
SIN	153
SGN	154
SQR	154
STOP	154
TAN	154
TIMEOUT	155
XOR	155
KEYBOARD SUMMARY	157
Editing keys	157
Accelerator keys	157
APPENDIX A - SCRIPT FILE GRAMMAR	161
Definition	161
APPENDIX B - SCRIPT FILE EXAMPLE	169
INDEX	177

Preface

About this guide

This guide explains how to use the Micropak 870C simulator based development tool.

The first section, **Introduction**, provides a rapid overview of the product and its intended uses.

The next section, **Installation**, explains the steps required to install Micropak 870C onto your system.

The **Fast Startup Guide** provides concise instructions on using the simulator for creating and debugging projects. This should be used to gain an overview in using and controlling the tool.

The **Tutorial** section takes you through, step by step, a sample debugging session and includes illustrations of some of the various ways the product can be used and the facilities it offers.

User Interface Details is a wide ranging section, covering the elements of the Micropak 870C screen display, the operation of the menu system, the tool bar and the other possibilities for controlling the simulation. This section also describes the operation of the on-line help facility.

The next sections explain the fundamental facilities in more detail and include **Navigating and Editing Project Files**, **Controlling Execution**, **Trace Buffering** and **Breakpoints**.

The **Port Simulation** section gives important details of the approach adopted for the simulation of port lines and discusses ways in which these can be used to establish and test the interactions of the target firmware under test with external hardware.

Micropak 870C enables the states of multiple aspects of the simulation to be viewed and monitored, many whilst the simulated code is executing. These facilities are explained in the **Viewing Simulated Objects** section of the user guide.

The next section, ***Source Level Debugging***, details the facilities offered to support this activity. Source may be written in C, C-like language or Assembler.

The final section, ***Using Script Files***, describes in detail the usage, scope and flexibility of the in-built script file processor, which can significantly increase the power of the test environment whilst also improving testing productivity. It is recommended that this section is studied carefully before planning a detailed testing or development programme, especially where repetitive or regression testing is intended.

Appendices

There are two appendices.

The first contains a definition of the **script file grammar**.

The second appendix contains the **script file example used in the tutorial**. This illustrates how the script file mechanism can be used to extend the simulation to mimic the behaviour of external hardware.

Introduction

The TLCS-870/C microcontroller family

The TLCS-870/C family of devices are tailor-made for very specific, high-volume applications and offer an extremely attractive cost-performance ratio.

The family includes a variety of devices with differing on-chip peripheral sets intended for differing application areas. All the devices feature an 8 bit wide data RAM and an 8 bit wide program ROM area.

Micropak 870C - An overview

The 'simulation engine'

Micropak 870C is an advanced simulator based tool for developing and testing applications for the Toshiba TLCS-870/C family of microcontrollers.

The simulation includes the full CPU core and registers, on-chip peripherals and I/O ports, and is performed totally in software in the host PC - no external hardware is required.

Application code to be investigated can be loaded into the software simulation and run, just as in hardware based environments.

Execution time assessments

The effective speed of the test code execution will vary with the speed of the host PC and with other factors such as the detailed composition of the target program. However, although the simulation does not therefore execute in real time, the execution time for each individual instruction is calculated and totalled by the simulation engine, so that detailed time-critical code sections can be assessed and detailed execution time measurements can be made.

Details of this execution time can be seen in the coverage window

Port simulation

The simulation allows an external ‘Thevenin’ equivalent network, consisting of a single external voltage generator and a single external series resistance, to be connected to each port line. Both the voltage and series resistance can be controlled via the user interface or the script file. Extending the simulation in this way allows you to investigate the interaction of the firmware under test with external hardware components such as switches, LEDs, etc. It also allows checks on the drive capabilities of the ports, and the use of pull-ups and so on.

Using Micropak 870C in conjunction with OTP devices

Interactions with complex hardware peripherals are difficult to emulate with the simulation alone, and in these situations one-time programmable devices may offer a realistic low-cost development route. In these cases the structure and basic behaviour of the code under scrutiny can be developed in the simulated environment, ready for subsequent trials in one-time programmable devices to confirm the correct operation of these more complex interactions.

The user interface

The user interface of the product adheres to the accepted conventions for Microsoft Windows applications, reducing to a minimum the overhead associated with learning to control the facilities provided. Extensive use is made of the graphical capabilities of the interface to provide a clear and attractive display. The ability to make significant events in the target system visible is considered to be a strong feature of the product and can boost debugging productivity considerably.

An in-built editor is included to support program development during debugging.

The program is intended to interact with the TLCS-870/C tool chain. This provides automatic links for code re-building, and includes facilities which allow the execution of the program to be monitored at source level.

The script file processor

The product includes a powerful 'script' file processor which can monitor and control events in the simulated target system according to control 'programs' written in the script language. Script files use a 'BASIC-like' syntax and can be used to mimic the behaviour of external devices or to set up, run and check the results of repetitive or regression tests.

Using Micropak 870C in quality and other formal testing regimes

In addition to its development facilities, the simulator can also be used for formal qualification or other quality testing. Powerful batch testing facilities are included, simplifying the execution and documentation of regression testing after product firmware changes.

TLCS-870/C devices supported by Micropak 870C

For a list of the TLCS-870/C family members currently supported by the Micropak 870C simulator, please see the README file on the distribution diskettes or contact your Toshiba representative. A list of supported devices is also given in the on-line help provided for the product.

Installation

This chapter explains the steps required to install the Micropak 870C software, both in terms of running the SETUP facility provided on the distribution diskette(s) and creating a suitable directory structure for your project files.

The important files

The following files are supplied on the distribution diskette(s):

- MP870C.EXE the main executable file
- MP870C.HLP the help file for the Micropak 870C program
- CPYOUTPT.EXE this program is used to pipe output from the compiler/linker to MP870C.EXE
- CPYOUTPT.PIF this enables DOS programs to run in the background (needed for re-building)
- README.TXT this describes installation instructions and the devices currently supported
- Tutorial files for running the tutorial detailed in this user manual

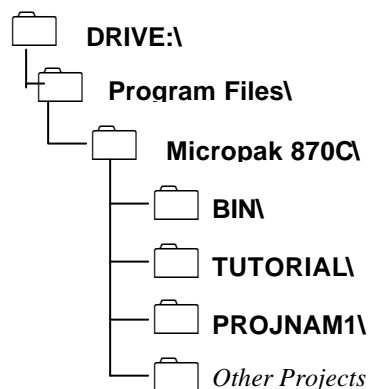
Note that you will also need the TLCS-870/C C compiler, C-like compiler or assembler, in order to generate object code to run and test.

Suggested directory structure

The suggested directory structure groups all the Micropak 870C related files under a main MP870C sub-directory but splits the Micropak 870C executable and help files from the source code and other project specific files.

Although other directory structures are possible, care must be taken to ensure that the run-time links to the compiler, linker and assembler from Micropak 870C will operate correctly. The mechanisms used, and a summary of the links which must be preserved, are described later in this chapter under the heading 'Run time links to the 870 tool chain'.

The following default directory structure is used by Micropak 870C:



The BIN sub-directory contains all the executable files, with the exception of the compiler executables. The TUTORIAL sub-directory contains all the tutorial files. Note that the tutorial files are not required for the normal operation of Micropak 870C.

Project specific directories

The sub-directories such as PROJNAM1 and PROJNAM2 are intended to hold project specific files, including the source files and the project batch file, which is used by the project re-building function to make the project. It is recommended that these sub-directories are given meaningful names to reflect the projects they represent, such as METER or TIMER, etc.

Running the SETUP utility

The distribution diskette contains a 'SETUP.EXE' installation program which is used to create an appropriate directory structure on the working drive and copy all the necessary files from the distribution diskette into the new directory structure.

The SETUP program must be run from within Microsoft Windows.

Run time links to the 870/C tool chain

Micropak 870C includes facilities for re-building and building complete executable code from the source using the standard components of the Toshiba 870 tool chain. These components are not installed as part of the Micropak 870C set-up and must be installed separately. Locate the tools in an appropriate directory (e.g. 'Toshiba') and edit the 'Autoexec.bat' file to add this directory to the 'path' entry.

Fast Startup Guide

This section gives an overview of the major facilities of the Micropak 870C simulator to enable the user to begin working with the tool.

Starting the Micropak 870C simulator

To start the Micropak 870C simulator locate the ‘Micropak 870C’ group window and double-click the ‘Micropak 870C’ program icon contained within it:



Micropak 870C

Setting up and debugging a new project

The Micropak 870C simulator provides a complete environment for writing, editing, assembling and testing your program. The environment for this is called the ‘Project’ and therefore the first step is to create a new ‘Project’ using the following steps:

- Create an appropriate directory for the project, either using ‘File Manager’ or the DOS ‘md’ command.
- Create a new project file by selecting ‘New’ from the ‘Project’ menu.
- In the ‘New Project’ dialog box, type the path and name of the project batch file (or click the ‘Browse’ button, locate the project directory and specify the name of the batch file). Note that the batch file must have a ‘.bat’ extension. Select the processor type from the list.
- Create the source files by selecting ‘New’ from the ‘File’ menu. Save the source files by selecting ‘Save As’ from the ‘File’ menu.

- Source files can subsequently be edited by selecting 'Open' from the 'File' menu.
- Invoke rebuilding of the source modules to create a runnable program by selecting 'Rebuild All' from the 'Project' menu.
- Create a test environment for the project including RAM displays, peripheral and port views, signal traces, setting breakpoints and building customised test panels, as desired.
- Debug the software using the various 'Debug' options including fast and slow run modes, setting the PC at a ROM address, tracing, etc.

Opening an existing project

Once a project has been created the project environment can be re-invoked at subsequent debugging sessions without the need to re-define it. The following steps should be used to open an existing project:

- Select 'Open' from the 'Project' menu.
- Continue with debugging the code, editing and re-building as necessary.

Common debugging features explained

How to set a breakpoint

Breakpoints can be set on program locations, memory read/writes and peripheral read/writes. Breakpoints are set by selecting 'Breakpoints' from the 'Debug' menu. There is also a toolbar icon for setting or clearing a breakpoint at the current cursor position.

How to start/stop execution

A set of execution control facilities are available from the 'Debug' menu. Execution starts from the current program counter position. The 'Reset' option, also available from the 'Debug' menu, resets the simulated processor or may be positioned anywhere in the code. Execution will terminate according to the option selected, e.g. at the current cursor position, or at the next instruction, or whenever a breakpoint is reached. A specific 'stop' instruction is also provided. All these facilities are available from the 'Debug' menu and as a collection of toolbar icons.

The time duration of an execution can be monitored using the 'Interval' window, which is displayed by selecting 'Interval' from the 'Window' menu.

How to edit source programs

Any text file may be created and edited using the options available from the 'File' and 'Edit' menus. Syntax colouring of C source programs can be applied if required so that, for example, comments and code are displayed in different colours. All normal editing functions are available, including 'Cut', 'Copy', 'Paste', 'Undo' and 'Find'.

How to navigate files

The 'View' menu option provides a number of facilities to help traverse active files, including jumping to a given line number. Bookmarks can be set in a file allowing the user to rapidly move to pre-defined points within their source files.

How to view data items

A RAM window can be displayed in order to view specified areas of RAM by selecting 'Device' from the 'Window' menu. Information about data items can be temporarily viewed using the 'QuickWatch' facility. Any data item displayed using 'QuickWatch' can be monitored throughout a simulation session in the 'Watch' window by selecting 'Add to Watch' from the 'Quick Watch'.

Information about the current local variables can be displayed by selecting 'Locals' from the 'Window' menu.

Information about the call stack can be displayed by selecting 'Show Call Stack' from the 'Debug' menu.

How to display peripherals and ports

The simulation of device peripherals and ports can be viewed by selecting the 'Device' from the 'Window' menu then choosing from the displayed list. One window will be displayed for each item selected.

Ports lines can also be displayed in signal windows. In this instance the port values will be shown as a 'scope-like' display. This facility can be selected by choosing 'Signal' from the 'Window' menu to open a signal window and selecting 'Signal Plots' from the 'Trace' menu.

How to build customised displays

Customised displays are built as test panels, where each panel can contain one or more items. The display of these items is controlled through the script file mechanism. In this case, the value of items such as script file flags and buttons are displayed to show the results of script file events. Bitmap images of components can be included within test panels to add realism to the project testing. Test panels are defined by selecting 'New Panel' from the 'File' menu.

How to create script files

Script files are created by selecting 'New' from the 'File' menu. These are text files containing 'BASIC-like' commands to control execution and to perform events at specified points during the execution. A full list of the script file commands is given in the chapter entitled 'Using Script Files'. Once a script file has been created it must run to assume control.

The appropriate script options, available from the 'Test' menu, are used to perform these tasks. Tool bar icons also exist for starting and stopping script file execution.

How to invoke tracing

A trace buffer is provided and can be switched on and off at random to capture required sections of executing code. This is achieved by selecting 'Debug' from the 'Options' menu and clicking the appropriate check box.

The buffer contents can be displayed using the 'Trace' and 'Debug' menu options.

How to configure the environment

The 'Options' menu and the 'View' menu contain a number of items which can be configured.

The 'Options' menu includes the following:

- 'Debug' options. Here you can select the run mode ('fast' or 'slow'), select trace buffering and enable signal output. You can also specify the size of the signal and trace buffers and the maximum number or time duration of script file instructions to be run for any one event.
- 'Editor' options. Here you can select the number of tab stops, the size of the 'undo' buffer and enable/disable the scroll bars.

The 'View' menu includes the following:

- Screen items. The tool and status bars can be turned on or off.
- Syntax colouring. Syntax colouring available for editing source files can be turned on or off.

How to set up automatic testing

Automatic testing can be achieved by writing one or more script files. Each test must be planned in detail and the correct test panels built to show the required output from the test.

How to obtain help

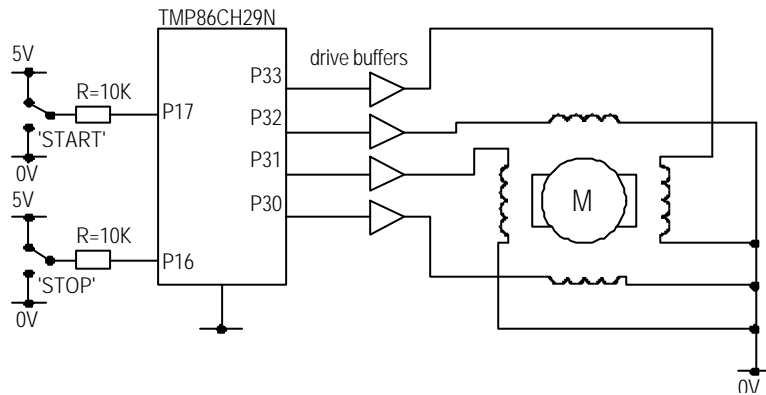
An on-line help facility is provided and is accessible by using the 'Help' menu option or by pressing the F1 key.

Tutorial

This section details an example Micropak 870C session, giving illustrations of the various ways in which the simulator can be used and the facilities it offers.

The example application - stepper motor control

The target code we use here is intended to control a stepper motor in response to two simple push-button control switches. The hardware environment is illustrated in this block diagram:



The simple stepper motor driver used as an example

A device will drive the 8 phases of a stepper motor via appropriate drive buffers.

Drive control is to be by two simple push buttons, each of which is wired so that operating the push button switches a port line between GND and V_{DD} . Port P1 pin 7 is used for the 'START' button and Port P1 pin 6 is used for the 'STOP' button. The individual motor phase coils are to be driven by 4 pins from Port P3.

What the example firmware needs to do

The firmware must generate the appropriate series of stepper motor drive signals.

It should also monitor the state of the push buttons to detect switch operations and interpret these so that one button shall cause the stepper drives to start, and the other shall cause the motor to stop. Only new switch presses should be actioned.

From the point of view of initialisation and preparation, the firmware will need to activate the output buffers for the stepper outputs, and set up an on-chip timer to trigger the timing of new phase drives. We will return for a more detailed look at the firmware later, but for the moment we must consider what is required for the test environment and how this environment is created using the Micropak 870C simulator.

Setting up the test environment

In order to exercise the controlling firmware it will be useful if we can set up Micropak 870C to mimic the effect of the two switches, and to monitor the phase drive outputs.

The switches and phase drives are the basic inputs and outputs of the controller and are therefore the minimum we need for 'black box' testing. However, it will probably also aid the debugging process if we can see the activity of the internal timer used for stepper phase timing and some of the important program variables. We will then be able to see the relation between the internal activity of the firmware under test and the actions it makes on external conditions.

Creating a new project

We start by creating a new project batch file for the firmware under test. Select the menu options 'Project', 'New', and then enter the name of the project file in the New Project dialog box i.e.:

'tutorial.bat'

At this point the processor to be simulated can be chosen from a selection list. You may however leave the default processor at this time.

Editing the project file

The next stage is to edit the project file in order to specify the source files for the project and the linker options. You should therefore select the menu option 'Project' 'Edit'. Files can then be selected from the file list for inclusion in the project. Each file is included by making your selection and then pressing the 'Add' button. The following files must be included:

1. stepper.c
2. io86xx29.c
3. startup.asm
4. stepper.lcf

Once these files have been added to the project you should press the 'Close' button.

Building the project

Now that the project contents has been specified you should instigate a project rebuild to generate the target code. This can be done by selecting the 'Project' 'Rebuild All' menu option or by using the following tool bar icon:



The rebuild icon

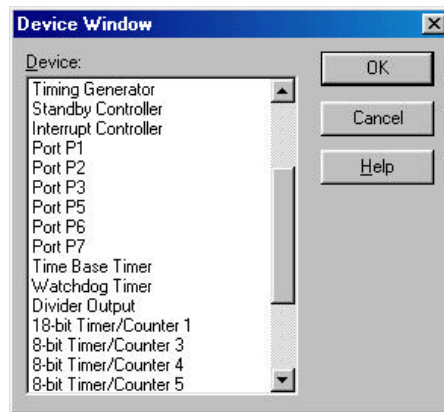
Output from the build tool chain will be shown in the Output window. When the re-building process is finished, the Output window should be closed and you will then see the source code displayed in a source window.

Displaying peripherals and data

Peripheral and data items for display may now be selected. To display the internal timer therefore select 'Device' from the 'Window' menu to obtain a list of the device windows available and select the 'Time Base Timer'. The illustration below shows the selection of the timer device.

Once opened any window may be moved, re-sized, maximised or minimised as in any standard Windows application. The second illustration below shows the timer window selected.

Here we are selecting a window for display:



Using the menus to select a timer view item

Here is the resulting device window:



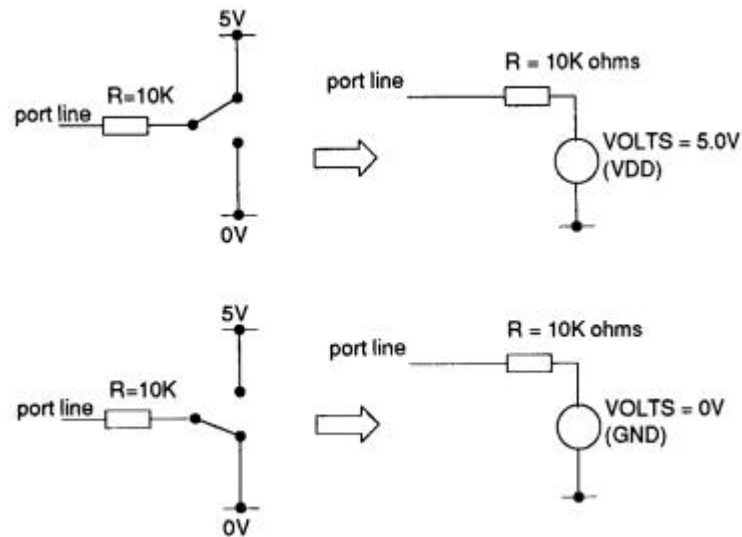
The view item for a timer

This window shows the states and activity of the internal timer we will be using. At this stage, (i.e. before executing any program sections) this will be in the 'reset' condition.

Setting up the push buttons

Setting up the switch simulations requires a little more thought. Micropak 870C includes, for each port line, the ability to simulate a single (perfect) external voltage generator in series with a single external series resistance. The voltage of the generator and the resistance of the series resistance can both be changed as required whilst the simulation is running. Furthermore a connection between the external circuitry and the port can be set or broken. For more information about this facility see the later section on *Port simulation techniques*.

For our purposes here, we can use this possibility to simulate the two conditions of the switches by changing the voltage generator from VDD to GND and vice versa. This can be seen as follows:



Minimising buttons - voltage changes

The following illustration shows the pin windows set up with required values. To enter values you merely over-type in the relevant boxes. The illustration shows the two connections in the V_{DD} condition.



The port pin windows show and set pin conditions

We will start with the switches in the V_{DD} condition. Select 'Pin' from the 'Window' menu to display a list of all possible pins. Then select 'P16' and 'P17' from the list to display these windows: now set the external network components (V_i) and (R_i) of each pin with the values shown above. Finally, click on the Connect check boxes of each pin to make the connections.

Phase output drive displays

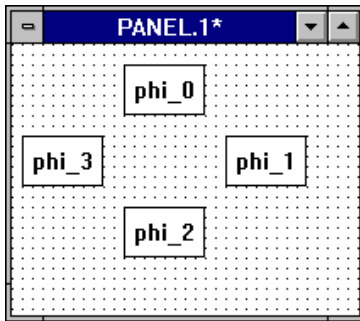
It will be interesting to monitor the drive outputs in two ways.

Firstly, we can set up a view in which the state of the outputs is represented as a circular pattern, as an analogue of the physical arrangement of the associated coils within the motor itself.

Secondly we can use the Micropak 870C plot recording style of displays in Signals windows to show the detailed timing relationships between the phased outputs.

Setting up a test panel of grouped items

Test panels containing groups of items for display can be specified. Firstly, select 'New Panel' from the 'File' menu to provide a new test panel in which to work. Now select 'Show Panel Palette' from the 'Test' menu to enable the items for the panel to be specified. A palette will be displayed showing the items you can display. For this example we will select a flag item for each of the individual Port P3 output lines. This is illustrated below:



Individual phases - inverse video means active

Item types, either buttons, flags, text or edit boxes are selected from the palette which is displayed when the edit mode is entered and these items can be placed in the test panel by 'clicking' the mouse at the required position in the panel.

You should select the 'flags' option by clicking the 'flag' radio button in the Palette window. You should then place four flags in the panel by moving the mouse to four different places, clicking on each place to set down one flag. The move option may then be selected to make any further adjustments to the positions of the flags.

The final task is to specify the properties of the flags. A properties dialog box can either be selected from the 'Test' menu or by double-clicking the flag when 'Move' is selected on the palette. The properties of each item consist of a unique caption to be displayed and an identity which is used to connect the item with script file control. You should give the names "phi_0" to "phi_3" as both the identity and the caption for each of the port lines respectively.

The visual state of each item (either active colour or background colour) is controlled by the actions of a script file which will need to be invoked in order for the test panel to be activated. The facilities provided by the script language are described in a later section. However, for our purposes we need to be aware that to use these facilities it is necessary to define a test panel containing the items for display and to set up a script file to define the conditions for controlling the display of the items. The properties dialog box allows you to specify the colours used in the display and the shape to be displayed. You may select different shapes or colours for the items to try this facility yourself.

Using the editor to define a script file

In order for the items in the test panel to display the status of the port lines, a script file needs to be written. Here we will use the editing facilities of Micropak 870C to write a simple script file for this task:

```
P60% = pin(60)
P61% = pin(61)
P62% = pin(62)
P63% = pin(63)

on event(p60%) run show
on event(p61%) run show
on event(p62%) run show
on event(p63%) run show

event(p60%) on
event(p61%) on
event(p62%) on
event(p63%) on

show:
  setflag "phi_0", getv(60) > 2.5
  setflag "phi_1", getv(61) > 2.5
  setflag "phi_2", getv(62) > 2.5
  setflag "phi_3", getv(63) > 2.5
end
```

A sample script file to input

Script files consist of one or more event handlers. The first initialisation event handler defines the conditions to be recognised and enables further event handlers to perform the tasks required when the relevant conditions are encountered during the simulation. In this instance our initialisation handler consists of four condition statements testing a value to be true or false, one for each of the phase output pins. The value tested is that evaluated from the actual pins as read using the previously defined 'pin' statements; the numbers (61, 62, 63, 64) correspond to real pin numbers on the device. All four condition statements are similar and define 'show' as the entry point of the handler to be used whenever the corresponding pin voltage is changed. During the initialisation, all four event conditions are enabled thus thereafter any change in voltage on any of these pins e.g. pin 61, will trigger the event handler 'show'. 'show' fetches simulated pin voltages for all four pins and updates the test panel flag items called 'phi_0', 'phi_1' etc., to show the current pin conditions. NB To view the pin numbers for specific pins select the 'Show Pin Number' option in the Pin dialog box

The screen display above shows the contents of the script file to be created. You should create a new file by selecting the 'New' option from the 'File' menu. You can then enter the text directly using the normal edit facilities and save the file when completed, giving the name:

PORT3.SCR

To see how the script file operates you must ensure that the script file window is activated then run the script selecting 'Run' from the 'Test' menu.

To check the operation of the script you can now enter some values directly into Port P3 and you will see the test panel display change according to the values entered.

To do this you should open the Port P3 window by selecting 'Device' from the 'Window' menu and choosing Port P3 from the list of devices. Firstly set the PCR to all 1's (output). If you now enter values into the 'P' box in the window you should see the display in the test panel change. Entering '0110' (binary for '6') will bring on two of the outputs and entering all '1's (binary for 'F') will bring all the outputs to the active conditions. What is happening here is that these actions trigger the script event handler 'show' which then updates the test panel.

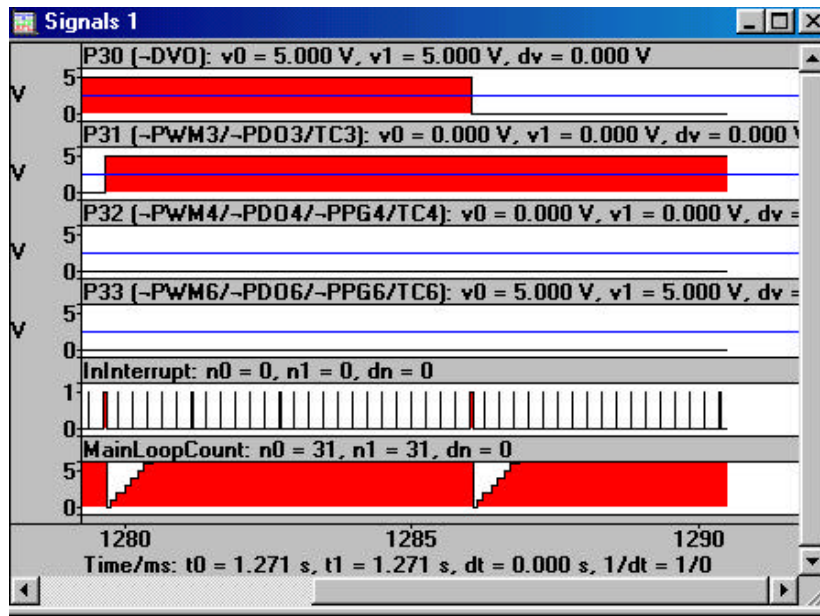
We can now move on to an alternative possibility for displaying port conditions.

Setting up a plot recording of the drive lines

Selecting 'Signals' from the 'Window' menu will display an empty Signals window. You must now specify the items you wish to plot by choosing 'Edit Signals' from the 'Trace' menu. Either pin signals or simple values read from data variables may be plotted. The default type shown is pins. The list of pins should be expanded and you should specify the individual lines, P30 – P33, clicking the 'Add' button after each selection. The result should be the inclusion of 4 Port P3 outputs within the one window. You may also wish to include the two significant data variables, Count and TimerTick. To include these change the 'Type' box to show L-Value and enter the names of the data variables in the L-Value box.

In order for the Signals window to plot the lines during processor execution we must enable the signal operation and ensure that the signal buffer is set to a size capable of holding enough information for plot records. To do this select 'Debug' from the 'Options' menu. A dialog box will appear, on which you should click the 'Signal Buffer Enabled' check box and enter a value of '1' (representing 1 second), as the size of the signal buffer.

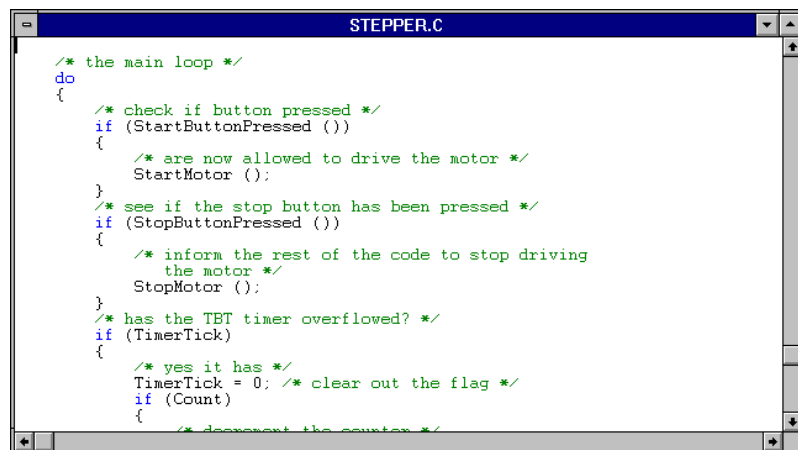
The Signals window will initially appear empty. The plots are only drawn as the simulation progresses. Here is an example of how the box might appear later when you have run the firmware:



Signals windows show the relative timing of signals clearly. They can be scrolled backwards to see the past changes. The amount of past detail that is stored depends on the buffer size. For more information about using plot records see the later section.

Loading the example firmware

The distribution disk contains the necessary source and other files to see the example running. Assuming that you opted to install the tutorial, these files will be in an appropriate directory on your system and you can load and run the sample program. In this instance the files will have automatically been loaded when you rebuilt the project and a source window displayed. If you have minimised this window you should now restore:



```
STEPPER.C
/* the main loop */
do
{
    /* check if button pressed */
    if (StartButtonPressed ())
    {
        /* are now allowed to drive the motor */
        StartMotor ();
    }
    /* see if the stop button has been pressed */
    if (StopButtonPressed ())
    {
        /* inform the rest of the code to stop driving
        the motor */
        StopMotor ();
    }
    /* has the TBT timer overflowed? */
    if (TimerTick)
    {
        /* yes it has */
        TimerTick = 0; /* clear out the flag */
        if (Count)
        {
            /* decrement the counter */

```

A sample source file

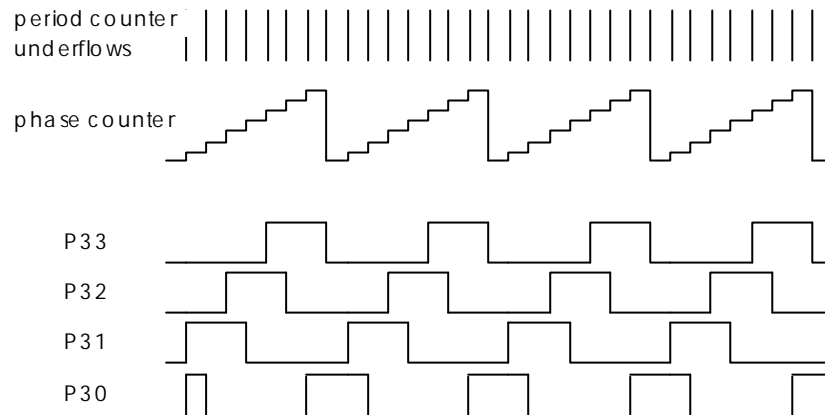
The source is displayed as it is held in the file. That is all user comments etc. are included in the display. If the source contained in the file is 'C' code then the normal display will show only this code. You can however request a disassembly of the loaded executable code to be interleaved with the relevant 'C' source by selecting the 'View' 'Show assembly' menu option.

The scroll bars, the cursor and PgUp, PgDn keys enable you also to scroll forwards and backwards through the example source code. Use these facilities to examine the example source code, and relate it to the descriptions which follow. The 'View' menu has options for setting and removing 'Bookmarks' in the window and moving between the marks set. Lines on which Bookmarks have been set are coloured.

Understanding the example

Timing

The main features of the system timing are shown below:



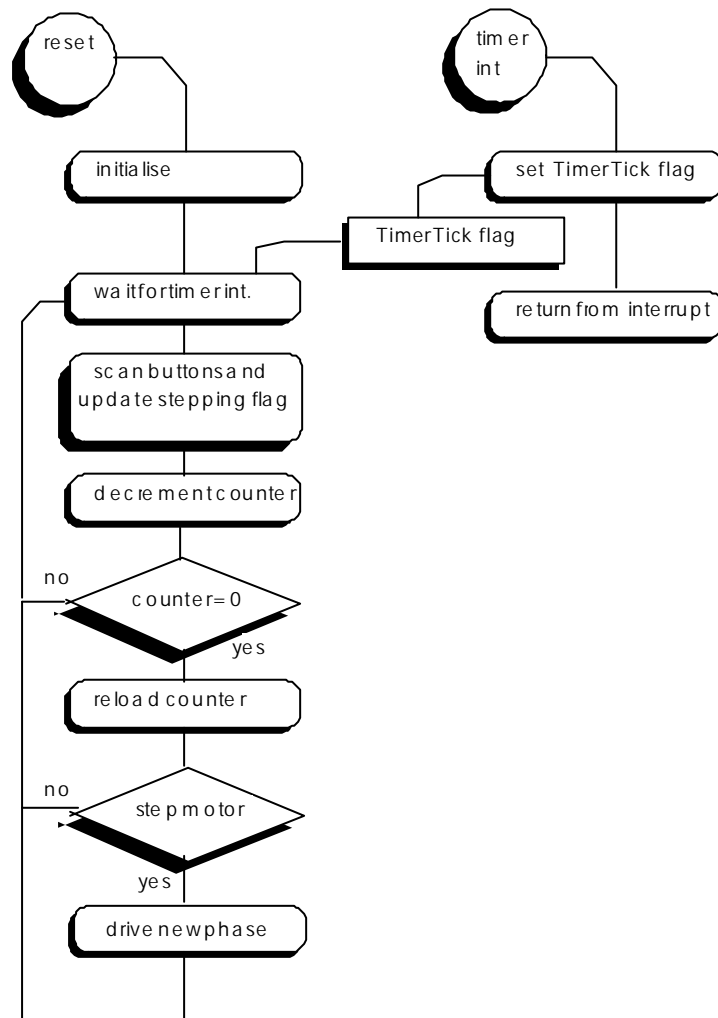
The timing of events in the example

The on-chip timer is used to generate a regular timing interrupt.

This interrupt causes a period counter to be decremented. When this counter reaches zero, a new phase pattern is driven to the motor circuitry and the period counter is reloaded to time the next inter-phase gap.

Program flow

The flow chart which follows shows how the program is structured to perform these various actions.



Control flow in the example

After initialisation, control enters the main loop.

Each pass round the loop is synchronised to the timer interrupt by waiting at the start of the loop until the flag 'TimerTick' is set by the timer interrupt handler.

The activities of the loop begin by scanning the 'start' and 'stop' buttons and setting or clearing the drive flag appropriately. Next, the period counter is decremented and tested. When there are no more ticks in this interphase period (i.e. the counter reaches 0) the counter is reloaded with the value of the current period. The drive flag is then tested to see if the motor is in operation. If it is, a new phase pattern is driven out to the motor.

'C' startup code

The code for this example is written in 'C'. The standard C compiler startup code (assembler) is used to provide the 'C' environment initialisation. This code is run at power on before control passes to the 'main' 'C' function.

The startup code begins at the label:

`_startup`

Initialisation

The initialisation function is run first. This sets up the program variables to appropriate initial states and sets up the interrupt controller and timer to generate a timer interrupt at pre-set intervals. Program variable set up includes setting the phase state to zero, and the counter, which counts down to the next phase, set to the maximum value ready to start counting down. The drive flag, indicating whether or not the motor is operational, is set to TRUE.

Finally, interrupts are enabled, and control passes back to the main loop.

Main code

The code consists of a continual loop. Firstly, when a new key press is detected, the motor is switched on or off accordingly. Then a test of the flag set by the timer interrupt is made. When this is detected, the switch lines are scanned by appropriate routines one by one.

The period counter ('Count') is decremented at each pass of the loop. When it reaches zero, it is reloaded with the constant 'Period'. The drive flag is then tested to determine if the motor is operational, and a routine is called to drive new phase outputs. Control then returns to the start of the loop.

```
do {
    unsigned char Temp;

    // check if start button pressed
    if (StartButtonPressed()) {
        // are now allowed to drive the motor
        StartMotor();
    }

    // see if the stop button has been pressed
    if (StopButtonPressed()) {
        // inform rest of the code to stop driving the motor
        StopMotor();
    }

    // drive the motor if necessary
    __asm(" di");        // disable interrupts first
    Temp = TimerTick;
    TimerTick = 0;
    __asm(" ei");        // now re-enable them
    if (Temp) {
        if (Count > 0) {
            Count--;    // decrement the counter
        }
        if (Count == 0) {
            Count = Period; // setup counter for next phase
            if (Stepping) {
                Phase++;
                if (Phase > 7) {    // check for overflow
                    Phase = 0;
                }
                SetMotorPort();    // drive the next phase
            }
        }
    }
}
while (1);                // loop for ever
```

The main line code

Processing functions

In the example, for clarity, the task is broken down into several functions. They perform the following:

- scanning for new button presses (one for each of the two buttons).
- getting a new phase into the motor drive outputs.
- setting a new phase into the motor drive outputs.

These individual routines are relatively straightforward and are not specifically listed here. You can inspect the source for these routines by scrolling the code displayed in the source window.

Interrupt handler

The timer interrupt is kept as simple as possible. The only significant action is the setting of the flag which the main line code tests to detect the passage of another interrupt period:

```
static void __regbank(2) IntTBT(void)
{
    // report to the main loop that the interrupt has occurred
    TimerTick = 1;
}
```

The handler for the timer interrupt

Executing the code - The PC indicator and cursor

We are now ready to try executing the code. First of all, generate a reset in the system by selecting 'Reset' from the 'Debug' menu. The PC will be set to the first line of the real program and is shown by a yellow bar. This will be in the assembler 'C' startup code source window at the label **_startup**.

Place the cursor in the 'stepper.c' source window and scroll down through the code. Move to the start of the 'C' **main** function

Click on the first line of code after the function declaration i.e. the call to the initialisation function. The cursor is set here.

Go to cursor

You can now give the 'Step to Cursor' command to execute the program to this point. This command can be given either by selecting 'Step to Cursor' from the 'Debug' menu or by clicking the appropriate tool bar icon:



The 'Step to Cursor' icon

When execution arrives at this point, you will notice the program counter (PC) indication by the changed colour at this line.

The source line, which corresponds to the current program counter position, is shown in yellow. Because the program counter points to the next instruction to be run, the yellow line is the one that is about to be run.

Single Stepping

It will be interesting to watch the effects of the code on the view items we have already set up. Single stepping gives you the opportunity to observe these individual effects clearly, instruction by instruction.

Single stepping can be done by selecting 'Step Into' from the 'Debug' menu or by clicking the appropriate tool bar icon:



The 'Step Into' icon

As execution proceeds you will see the yellow bar showing the current program counter position gradually move through the routine.

Notice that because the program counter points to the next instruction to be executed, the PC indication shows the actions that are about to be performed, not the action that has just been carried out.

Animating the code - multi-stepping

Multiple stepping executes one step at a time, and displays the program counter position after each instruction by moving the yellow bar. This function, sometimes known as 'animation', shows execution moving through the code, making program flow clear.

Animation can be started either by selecting 'Go' from the 'Debug' menu or by clicking the appropriate tool bar icon:



The 'Go' icon

Note, however, that because the whole screen display is updated, including re-writing the source window and showing the program counter bar, execution speed in 'animation' mode can become slow. When using this mode of execution speed may be increased by minimising those windows you are not interested in observing at this time.

Fast debug mode

Having observed the individual stages of initialisation we can now proceed to run the code in the fast debug mode. In this mode, only the test panels are updated during execution and therefore the speed of execution is increased. To initiate this mode either select 'Fast' from the 'Mode' field of the 'Debug Options' dialog box, having selected the latter from the 'Options' menu, or click the appropriate tool bar icon:



The slow icon (A tortoise) The fast icon (A hare)

Running in fast mode should enable you to see the stepper drive activity as a circulating effect in the test panel display.

Simulating button presses

With the code running, we can now investigate the behaviour of the firmware in response to button presses. For this you should return to the slow debugging mode, so click on the debugging mode icon to return to slow mode. The icon shown represents the current state, so you should click on the 'hare' to show the 'tortoise'.

You can simulate the effect of a push-button operation by changing the voltage generator from 5v to 0v. This mimics the effects of the switch operation causing the voltage at that pin to fall, triggering the main loop to detect a switch operation and stop the motor. To simulate the stop button therefore you should restore or make active the P17 pin window and change the voltage appropriately. To release the button you would change the voltage back to 5v again, however, you should pause for a short while between the operations to ensure that the switch sensing code sections are run.

To start the motor again you should simulate a button press on pin P16. The results of each button press should be seen in both the test panel display and the plots recorded in the Signals window.

Project context files

So far we have seen how to set up a test environment and use it to run our target code. You may find it useful to save this environment in order to re-instantiate the test at a later stage. To save your panel file choose the 'File' 'Save As' option. The file should be saved with the extension .PAN. The context information including information on the windows and panels that were open and their position may be saved through the 'Test', 'Save As' menu option. You must give the test context a file name when saving it. If you wish the context to be opened automatically when you open a project you should give the file the same name as the project name. The extension is .TST.

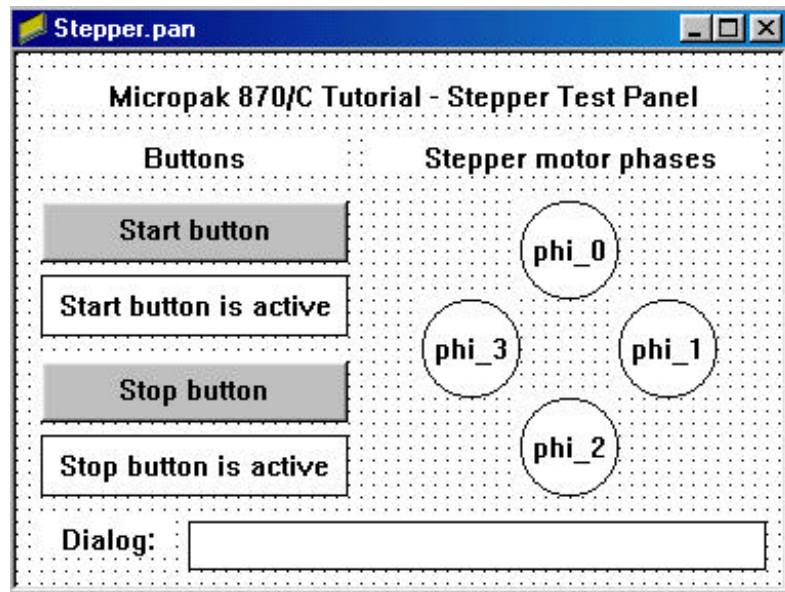
When we started the tutorial we did not open an existing project file, as the tutorial was described to create a new one. An example project file for the tutorial is provided to go with the tutorial example:

stepper.bat

To invoke this use the 'Project', 'Open' menu sequence. You will then see a pre-defined set of windows appear on the display. The next section describes how test simulations may be made easier and faster by using script files associated with the pre-defined project context file.

A new window displaying two 'Window style' buttons, labelled as 'STOP' and 'START' will be seen. These items are linked via the script file to the two button sensing pins.

This is how the new window will appear:



The test file reads in the test panel

Using the script file

Micropak 870C includes a 'script file' processor as described briefly in an earlier section. This allows events in the simulated environment to be controlled and monitored automatically according to details defined in the script file. Script files are described more fully in a later section, but we will illustrate their use here by providing a more convenient way of running our test session.

The distribution disk contains a sample script file to go with the tutorial example:

stepper.scr

To invoke this, select 'Open' from the 'File' menu. This will display a dialog box allowing the file to be loaded. Select or enter the name ('stepper.scr') and the script file itself will appear in a 'text window'.

To action the script in the file, select the Script file window and then use the menu option 'Run Script' from the 'Test' menu to action the file commands. The results of any further simulation will then cause changes to the display according to the events translated by the script file.

Clicking on the button items shown in the new test panel will toggle the voltage switch on pins P16 and P17 to simulate the button press and release.

For more details see the later section on script files.

Checking the generated sequence

If you study the generated output phase signals carefully you may notice an anomaly. In fact, the example program includes a bug, which results in one of the patterns driven out to the port being incorrect and showing all the lines being driven high together. This will result in a disturbance to the circulating pattern and can also be seen in the plot record display in the Signals window.

Tracking down the bug

In order to find the problem we will set a breakpoint just after the code line which sets new phase patterns into the hardware.

We will also activate trace buffering so that we can ‘back-step’ from the breakpoint, to see the code line which generates the incorrect pattern.

Firstly, we must stop our current execution. To do this use the ‘Alt-F5’ key or select ‘Stop Debugging’ from the ‘Debug’ menu.

Setting a breakpoint

Scroll through the code until you find the function called:

SetMotorPort

In this routine the line:

```
motor = GetMotorPortDriveForPhase();
```

is the line which writes to the hardware drive port. Click on the code line just after this to set the cursor position. This line has the instruction:

```
return;
```

To set the breakpoint here click the ‘Toggle Breakpoint’ icon:



The ‘Toggle Breakpoint’ icon

Lines on which breakpoints are set are shown in red. Clicking the breakpoint icon toggles the breakpoint at the cursor position on or off.

Once we have set the breakpoint we need to turn on trace buffering so that on reaching the breakpoint we will be able to step backwards and check the previous program actions.

Activating trace buffering

To activate the trace buffer, select ‘Debug’ from the ‘Options’ menu and click the ‘Enabled’ check box for the trace buffer. Micropak 870C allows the size of the trace buffer to be adjusted, but the default size will be suitable for us here.

Running to the breakpoint

Now we have set the breakpoint and activated the trace buffer we can run the code again (the 'Go/Stop' icon is probably the easiest way). When the breakpoint is reached execution is halted, a message appears and a beep is heard.

You will notice that the yellow line indicating the PC marker is now on our chosen instruction, and that a new pattern will just have been driven out.

Checking the individual drive patterns

You should now restart the execution using the 'Go' icon or by selecting 'Go' from the 'Debug' menu. Correct patterns should activate one output or two outputs simultaneously. If the current state of the phase output.

shown in the panel window currently has all outputs activated, the erroneous pattern must have been driven out. If the current pattern appears correct (i.e. it has one or two outputs active), then you can run to the next pattern by using the 'Go' icon again. Continue this process until the faulty pattern has just been driven out.

At this stage we know that the pattern just output was wrong. Examining the code shows that the pattern is written to the data item '**motor**' from values derived in a '**switch**' statement in '**GetMotorPortDriveForPhase**'.

The '**switch**' statement contains each of the specific patterns written to the data item '**motor**'. In order to know which of the '**switch**' '**cases**' set the faulty pattern we need to know which lines were run just prior to the hardware updating.

The 'Back-step' facility allows us to 'turn the clock back' and effectively run the code in reverse, tracing the execution back up through the code.

To invoke this facility select 'Step Back Into' from the 'Trace' menu.

Using this facility you will find that the following statement was the villain:

case 3:

return 0x0f;

The pattern in question is specified here as a hexadecimal constant (0x0f) and it can now be seen that this does indeed set all lines active. In fact this value should be 0x06. Having located the bug we can now move on to see how to make an appropriate correction.

Making corrections

To correct this bug you can use the source edit facilities of Micropak 870C directly.

Position the cursor in the faulty line and make the correction using the cursor keys and over-typing, etc. After editing you must then save the file.

Re-running after corrections

You must now rebuild the project to ensure that the corrections are included in the simulation. To do this select 'Rebuild All' from the 'Project' menu or use the tool bar icon as described in an earlier paragraph..

The Toshiba tool chain will be automatically invoked and a new executable file will be generated. Once the rebuild process has finished, the project files will be re-loaded and a refreshed source window will be displayed.

You should now be able to re-run the code (e.g. by clicking the 'Go' icon), and this time the phase output sequence should be correct.

Moving on

Having been through this tutorial session you should have some idea of the facilities which Micropak 870C offers, and how these can be used to good advantage in your own testing or development programme.

The later sections of this guide provide reference information covering all of the various facilities in more detail.

Perhaps before leaving the tutorial set up, you might find it helpful to use the context of the tutorial example to experiment with some of the other facilities described in later sections. Here are some suggestions:

- **Step Over** - to skip over, say, the button testing routines.

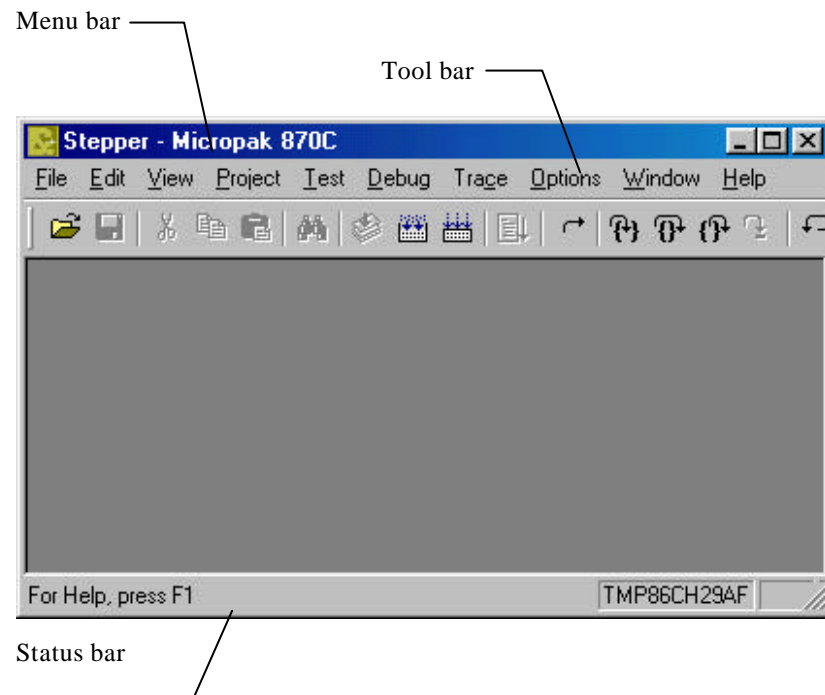
- **Step Out** - single step into the 'GetMotorPortDriveForPhase' routine and then use the 'Step Out' function to run to the end of this routine. This facility is one of the most useful when investigating code so it's beneficial to learn how to use it early on.
- **Watch windows** - Micropak 870C provides various ways of viewing data variables. Try adding some data items to a Watch window to show the interphase counter ('Count') and the tick recording variable ('TimerTick').
- **Trace buffering** - enable the trace buffer for a while, observe the effect on speed, then use the 'Back-step' facilities to watch the code running in reverse. Note that the PC is shown in green.
- **Port views** - ports can be viewed multiple lines at a time. Showing Port P0 in this way would be a good example of the use of this facility.
- **Breakpoints** - use the 'Breakpoints' menu option to set a breakpoint at the code line which deals with the 'Start' button being detected as active, then run the code, activate the button and see if the breakpoint is correctly reached.
- **Go and Go** - set a 'Breakpoint' at the code line which drives a new phase of the motor i.e. the function call to SetMotorPort. Then select the Go and Go option from the Debug menu whilst operating in 'Fast Mode'. The code will run in fast mode and when it hits the specified breakpoint Micropak 870C will update all windows before starting to run the code again. Take note of the Signals window updates.
- **Source editing** - try some of the editor functions available on the 'Edit' menu.

You will probably find most of these functions can be invoked easily using the menu system. However, if you need more explanation or information, try the on-line help facility, or the later sections of this manual.

User Interface Details

The Micropak 870C screen

When the Micropak 870C program is started the following window is displayed:



The main user interface elements are as follows:

Name	Description
Menu Bar	Lists the available menus, e.g.:

File Edit View Project

Menu	When a menu is selected it lists the commands specific to that menu, e.g.:
-------------	--

Debug	Trace	Options	Wi
Go		F5	
Step Into		F6	
Step Over		F7	
Step Out		F8	
Step to Cursor		F9	

Note that when a menu option is not available it is displayed 'greyed out' and cannot be selected.

Tool Bar	The tool bar displays a number of buttons which provide quick access to some of the menu commands e.g.:
-----------------	---



Desktop	This is the background area of the screen.
----------------	--



Icon	This displays a window in a compact form, e.g.:
-------------	---



Port P6

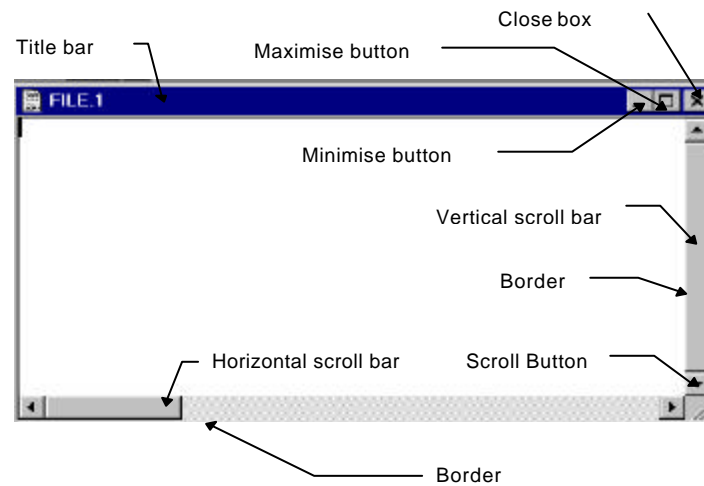
Window	This displays source code, help information, simulated objects etc., e.g.:
---------------	--









Name	Description
Scroll Bars	These change the position within a file or list, e.g.: 
Status Bar	This displays information about your current session, including processor type, position within a file, etc., e.g.: 

Window elements

Note that the example given here is from Windows 3.1. Windows 95 elements which are different are shown as second alternatives in the table following this screen.



Name	Description
Window border	Used to size the window by dragging with the mouse
Window title	Indicates the window contents. Also used for moving the window by clicking and dragging
Close button	Closes the window when double-clicked, e.g.: 
Minimise button	Reduces the window to an icon when clicked, e.g.: 
Maximise button	Enlarges the window to its maximum size when clicked, e.g.: 
Restore button	Restores the window to its original size when clicked with the mouse, e.g.: 
Scroll Up arrow	Click once with the mouse to scroll up one line at a time, e.g.: 
Scroll Down arrow	Click once with the mouse to scroll down one line at a time, e.g.: 
Page Up area	Scrolls up one page at a time when clicked once
Page Down area	Scrolls down one page at a time when clicked once
Scroll button	Indicates the relative position in the file/list. Drag with the mouse to change position in the file/list

Menu operation

Menus can be invoked either by the appropriate keyboard actions or by clicking with the mouse. Menu options are grouped under main headings in a menu bar located across the top of the window.

Clicking on one of the options in the menu bar will cause the appropriate pull-down menu to be displayed, e.g.:



The options presented can then be selected by moving the mouse cursor to the desired option and clicking again. Note that menu options which are not relevant to the current operation are shown 'greyed out'.

Sub-menu options normally have a single character underlined. If the associated key is pressed whilst holding down the Alt key the appropriate sub-menu will be displayed. This provides an alternative to using the mouse to display the desired sub-menu.

In addition to this underlined character, some menu options also include a short cut key or accelerator key which can be used to invoke a function directly. These are described to the right of the menu text, e.g.: using the combination of the Ctrl and 'D' key will display the 'Device Window'.

Menu options which, when selected, display dialog boxes are shown with '...' at the end of the option field, e.g.:

Open...

Menu function reference descriptions

The following tables show how the functions accessed via the menu system are grouped:

File	Edit	View	Project	Test
New	Undo	Line	New	Open
Open	Redo	Show Assembly	Open	Save As
Close	Cut	Show All Assembly	Add Files	Run Script
Save	Copy	Next Error	Close	Stop Script
Save As	Paste	Previous Error	Compile File	Stop All Scripts
New Panel	Delete	Toggle Bookmark	Build	Show Panel Palette
Print	Find	Next Bookmark	Rebuild All	Hide Panel Properties
Print Preview	Replace	Previous Bookmark	Stop Build	Panel Properties
Print Setup	Read Only	Clear All Bookmarks	Configurations	Panel Grid Settings
Exit		Toolbar	Set Active Configuration	Recent Tests
Recent Files		Status Bar	Settings	
		Syntax Colouring	Recent Projects	

Debug	Trace	Options	Window	Help
Go	Go Back	Debug	New Window	Micropak Workbench
Step Into	Step Back Into	Editor	Cascade	Build Tools
Step Over	Step Back Over	Device Info	Tile	Device Info
Step Out	Step Back Out		Arrange Icons	About Micropak 870C
Step to Cursor	Step Back to Cursor		Stack Source Windows	
Go and Go	Clear Interval		Interval	
Stop Debugging	Edit Signals		Signals	
Reset	Signals Zoom In		Performance Analysis	
Set PC to Cursor	Signals Zoom Out		Watch	
Show Call Stack	Snap Signals		Locals	
Breakpoints	Clear Performance Analysis		Registers	
QuickWatch	Code Coverage		Output	
	Clear Coverage		Files	
			Browser	
			Device	
			Pin	

Menu : 'File'

This group of menu options deals with file handling and printing, including creating a new panel. Note that a list of the four most recently used files is maintained for convenience. The exit command for the Micropak 870C program is also accessed from this menu.

Option	Description
New	Start a new file
Open...	Open an existing file
Close	Close a currently open file
Save	Update the disk copy of the current file
Save As...	Write the current file to disk, optionally with a different name
New Panel	Create a new test panel
Print...	Print the current file
Print Preview	See on screen how the current file will appear when printed
Print Setup...	Review printing options
Exit	Leave the Micropak 870C program
Recent Files	Open a recently accessed file

Menu : 'Edit'

This group of menu options deals with the editing of files. Any text file may be edited using these functions. If a file has not been opened prior to editing, an 'untitled' file will be created which may be saved later. See the 'Options' menu for options relating to this menu.

Option	Description
Undo	Undo the most recent editing action
Redo	Redo the most current editing action
Cut	Remove the selected text and place in the clipboard
Copy	Copy the selected text to the clipboard
Paste	Insert the current clipboard contents
Delete	Delete the selected text from the current file or list
Find...	Find a string of characters in the current file.
Replace...	Find and replace a string of text with another
Read Only	Mark the current file as read only

Menu : 'View'

This menu deals with viewing the source text windows. Facilities are provided to place bookmarks in the source

Option	Description
Line	Go to a line number
Show Assembly	Toggle the current source file between normal and mixed source/assembly display
Show All Assembly	Toggle source files between normal and mixed source/assembly display
Next Error	Go to the source line containing the next error
Previous Error	Go to the source line containing the previous error
Toggle Bookmark	Turn bookmark on or off
Next Bookmark	Move to the next bookmark
Previous Bookmark	Move to the previous bookmark
Clear All Bookmarks	Clear all the defined bookmarks
Toolbar	Toggle the toolbar on/off
Status Bar	Toggle the status bar on/off
Syntax Colouring	Enable or disable syntax colouring

Menu : 'Project'

This group of menu options deals with project-wide facilities. This includes project context files, script files and rebuilding the executable file. See the 'Options' menu for options relating to this menu.

Option	Description
New...	Create a new project file
Open...	Open an existing project file
Add Files...	Add files listed for the project
Close	Close the current project file
Compile File	Compile an individual source file
Build	Compile changed files and link the object files in the current project
Rebuild All	Rebuild a project from scratch
Stop Build	Stop the current project rebuild
Configurations	Define build configurations for the project
Set Configuration	Select the configuration to be used when building

Settings	Set up all setting for the project configuration
Recent Projects	Load a recently accessed project

Menu : ‘Test’

This group of menu options handles the test script files and the settings of the test panel.

Option	Description
Open...	Open a test file
Save As...	Save the current window positions and debug options as a new test file
Run Script	Action commands in the selected script file
Stop Script	Stop execution of the selected script file
Stop All Scripts	Stop execution of all script files
Show Panel Palette	Display the panel palette, allowing you to add and edit items
Panel Properties...	Specify the properties of a test panel item
Panel Grid Settings...	Specify the grid settings in the test panel window
Recent Tests	Display the four most recently opened test files

Menu : ‘Debug’

This group of menu options controls the running of the program when debugging. It contains various run modes and allows breakpoints to be set. See the ‘Options’ menu for options relating to this menu.

Option	Description
Go	Run the target processor
Step Into	Run a single instruction
Step Over	Run a single instruction or procedure
Step Out	Run to a return
Step to Cursor	Run until the cursor position is reached
Go and Go	Run until a breakpoint is reached and run again
Stop Debugging	Halt the target processor
Reset	Generate a ‘reset’ in the target processor
Set PC to Cursor	Set the PC to the current cursor position
Show Call Stack...	Display the call stack dialog box

Breakpoints...	Set breakpoints
QuickWatch...	Inspect the value of the selected variable

Menu: 'Trace'

This group of menu options controls the trace buffer facilities and the signal plot display facilities. This menu also includes the facility to clear the interval window.

Option	Description
Go Back	Trace back to start of the buffer
Step Back Into	Step back one instruction
Step Back Over	Step back one instruction or procedure
Step Back Out	Step back to start of the procedure
Step Back to Cursor	Step back to the current cursor position
Clear Interval	Clear the interval counter
Edit Signals...	Add or edit plots in the signal box
Signals Zoom In	Magnify the signal box
Signals Zoom Out	Reduce the contents of the signal box trace
Snap Signals	Snap the signal plot lines to nearest value transition
Clear Performance Analysis	Clear the values shown in the performance analysis window
Code Coverage	Turn on the code coverage option
Clear Coverage	Reset the code coverage information

Menu: 'Options'

This group of menu options allows the user to specify options which control other menu facilities.

Option	Description
Debug...	Change options for the debug menu including trace buffer on/off setting
Editor...	Specify the options for the editor
Device Info	Specify the directory path for the processor data sheet.

Menu: 'Window'

This group of menu options allows the user to specify new windows to be displayed and rearrange existing windows.

Option	Description
New Window	Generate a new 'copy' of the current window
Cascade	Arrange all the open windows in a 'cascaded' display
Tile	Arrange all the open windows as 'tiles'
Arrange Icons	Tidy the display of the icons
Stack Source Windows	Arrange the open source windows in a Z order
Interval	Open a new Interval window
Signals	Create a Signals window
Performance Analysis	Open the Performance analysis window
Watch	Open the Watch window
Locals	Open the Locals window
Registers	Open the Registers window
Output	Open the Output window
Files	Opens the File window
Browser	Opens the Browser window
Device...	Open a new device window including RAM and Ports
Pin...	Open a pin window

Menu: 'Help'

This group of menu options gives the user access to the help facilities.

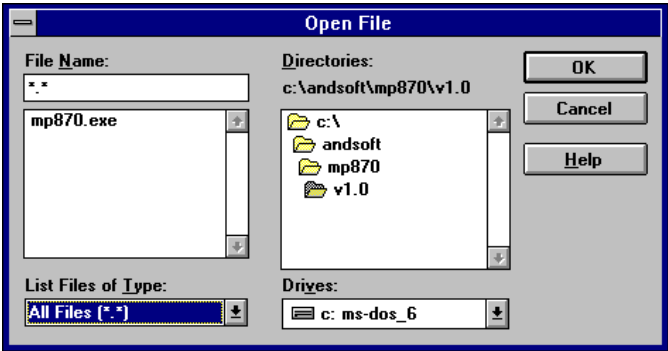
Option	Description
Micropak Workbench	Display help specific to Micropak 870C
Build Tools	Display help on the related TLCS-870/C tools including error messages and option settings.
Device Info	Access technical information about 870 devices
About Micropak 870C	Display version and copyright information

Using dialog boxes

Dialog boxes are used to enter values and make selections. Menu options which invoke dialog boxes are shown with ... adjacent to the menu text, e.g.:

Open...

Examples of dialog boxes which are specific to Micropak 870C are shown in the relevant section. Common dialog boxes such as those for opening files follow standard conventions. An example is given here:



In order to enter a value into a field within a dialog box you will need to click into the field using the mouse. The text cursor will then appear in this field, indicating where the text you type will appear. The following keys can be used within dialog boxes with their normal editing functions:

Key	Function
Ins	Toggle in and out of insert mode
Del	Delete the character to the right of the cursor
Left, Right, Up or Down Arrow	Non destructive cursor movement
Tab	Move to the next field on the window
Backspace	Delete the character to the left of the cursor

List boxes

These are special fields which allow the selection of one item from a list, and incorporate an arrowed button alongside enabling you to ‘pick and choose’ from the displayed choices, e.g.:



Radio buttons

Again, special fields, called ‘radio buttons’ require you choose between several choices displayed on the screen by clicking with the mouse in the circular area to the left of the field text, e.g.:



Check boxes

These fields enable you select or deselect an option by clicking the box with the mouse, e.g.:













Keyboard actions














Micropak 870C includes a set of pre-defined ‘accelerator’ key functions. These allow fast access to the most commonly used menu options. Where key functions exist they are listed alongside the menu options. A full listing of these is given in the ‘Keyboard Summary’ chapter of the manual.

The tool bar

The tool bar provides a quick and easy way to access most frequently used functions. The row of buttons, each with an icon representing the action which will be performed when the button is clicked using the mouse.

The tool bar icons

	Open Document		Step Over
	Save Document		Step Out
	Cut Selected Text to Clipboard		Step to Cursor
	Copy Selected Text to Clipboard		Reset the Processor
	Paste Contents Of Clipboard		Set PC to Cursor

	Find		Toggle Breakpoint at Cursor
	Compile File		QuickWatch
	Build		Clear Interval
	Rebuild All Project		Fast/Slow Debug Mode
	Run/Stop Script File		Open Device Window
	Go/Stop		Open Pin Window
	Step Into		

Using the on-line help system

Micropak 870C includes an on-line context sensitive help facility which can be invoked using any of the following methods:

- Pressing the F1 function key
- Using the Help pull-down menu
- Pressing the Help button on the dialog boxes

The help system consists of an index of a set of topics. They are shown in green. Topic names which are underlined show the user that a lower level sub-topic list will be displayed if this topic is selected. Where a topic is not underlined, the help system will display help text.

The user can select the topic required by clicking on a chosen topic name. Sub-topics or related topics, also shown in green, will be displayed as relevant and the user may select these by clicking on the item displayed.

Searches on any topic may be invoked by the user by selecting the help system search option. Searches are only available on indexed items, i.e. topics. You cannot search the help text itself.

A history of the help requested is kept and can be shown by selecting the History tool and the help requested may be back-tracked or printed.

To exit from the help system you should select the close tool or exit option from the help system file menu.

F1 function key

The F1 function key can be used to invoke help about any menu option or menu item. The user must highlight the option or item required and then press F1. The help system will then display help about the selected item.

Help on tools may also be invoked using the F1 key. Here the user must select the tool required by pointing the mouse at the tool and hit F1 at the same time.

Help menu

The help system may be invoked by selecting any one of the items on the help menu. An index of topics relating to the menu item selected will then be displayed. The user can then select the topic required.

If the search option is selected, the user will be prompted for the input of a topic name. A selection list of topics will be shown according to the information input.

Help option

The help option in the dialog box or on the menus may be selected to display help text about the item. Once displayed the help system remains invoked and the user may then search on other or related topics.

Device Information

This menu item is activated when the focus is set on a pin or peripheral window. Technical data from the standard Toshiba 870 data sheets held on CD will then be displayed for the device item selected.

In order for the data to be displayed the CD must be resident in the CD drive.

Navigating Project Files

Project file overview

Micropak 870C requires files to be organised by project. This section describes the assumed file grouping and explains how Micropak 870C uses the various files involved.

Project files

There are a number of files, each having a specific function, which collectively constitute a project. These files are as follows:

.BAT	Batch file used to control project rebuilding
.PRJ	Project information file
.TST	Window configuration details
.ABS	File containing code and debugging information from the compiler/assembler

Projects files are identified by the '.BAT' extension and are opened by selecting 'Open' from the 'Project' menu in order to set up a test environment. Please be aware that opening the project file does not automatically run script files. The source file will however be opened automatically, ready for editing or actioning.

The project file is a text file and can be edited directly. It can be used in one of two ways.

Firstly, if the project is 'internal' the batch file will contain just the text 'REM MicroPak generated batch file - Do Not Modify.' This will allow Micropak 870C to handle all the project rebuilding, allowing the user to add and delete source files and select from a comprehensive range of options. If you intend to use this 'internal' project rebuilding facility then do not delete or change the contents of this file.

The second way in which the project batch file may be used is for 'external' project rebuilding. If this option is desired, simply place in this file all the commands required for the project rebuilding sequence. Note that Micropak 870C does not perform any checking of the contents of this batch file - it will simply execute whatever commands are placed within it.

Source files are required for rebuilding

Micropak 870C does not require the original source files in order to execute code. However, it does require them in order to allow you to edit the source and to subsequently rebuild the executable file. It also requires the source file to enable debugging to take place.

Source files are text files and can be edited using the Micropak 870C editing functions.

The processor information file

The processor information file must be present within the project environment. It has the same name as the project but has a .PRJ file extension, e.g. 'PROJECT1.PRJ'. For both types of project (i.e. 'internal' and 'external') this file defines the processor type selected and the clock frequencies. In addition, for internal projects, this file also contains details of the source files and associated options.

The window configuration file

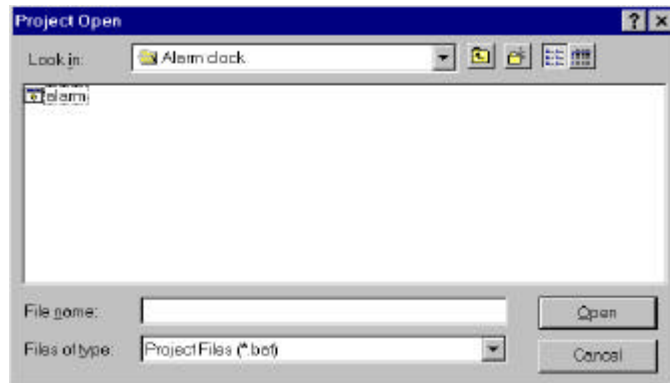
The window configuration file details your test environment in terms of window position information, breakpoints, watch item names, etc. It avoids the need to redefine your testing environment each time you start a new testing session. The file has the same name as the project but has a .TST file extension, e.g. 'PROJECT1.TST'.

The debug information file

This file, with a .ABS file extension, holds the code to be executed plus the debug information which is output from the re-build tool chain. This file is created as a result of the build process and normally uses the same name as the rest of the project, e.g. 'PROJECT1.ABS'. Note that if the project is 'external', an alternative file name may be used.

Opening a project

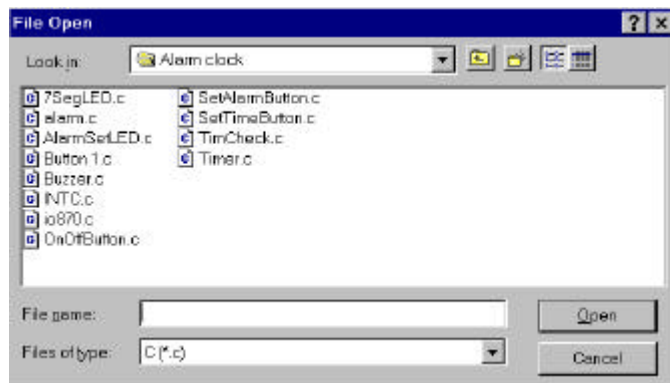
A project is opened by selecting 'Open' from the 'Project' menu. A dialog box similar to the following is displayed:



Browse through your drives and directories until the correct project batch file is located, select it, then click the 'OK' button.

The appropriate source file (with a .C, .CL or .ASM extension) will be loaded and displayed within a window on the 'desktop space' of MP870.

Further files (such as other source files and script files) may be opened by selecting 'Open' from the 'File' menu. The following window is displayed, allowing you to select the file required:



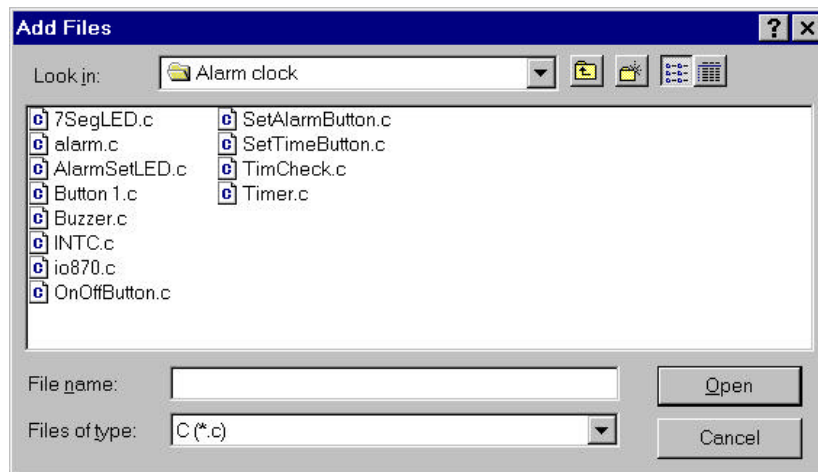
Files Window

The 'Files' window can be turned on by selecting 'Files' from the 'Windows' menu. This window will show all source and header files currently held for the project.



Editing a project

MP870C includes the facility to allow you to add and remove entries from the list of files which comprise a project. To add files to a project the 'Add Files' option is selected from the 'Project' menu and window similar to the following is displayed:



To add a file to the list of files within the project, click on the file names required and select 'Open'. The new file names including associated header files will then be seen in the Files Window.

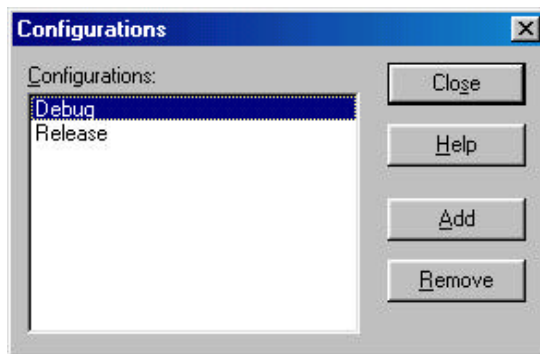
To remove a file from the list of files in the project select the file from the File Window and press the 'DELETE' key.

Specifying project settings

For both internal and external projects there are a number of options which you can specify, including processor type, build mode and command strings for rebuilding the project. Any specific set of project options selected is named as a particular configuration.

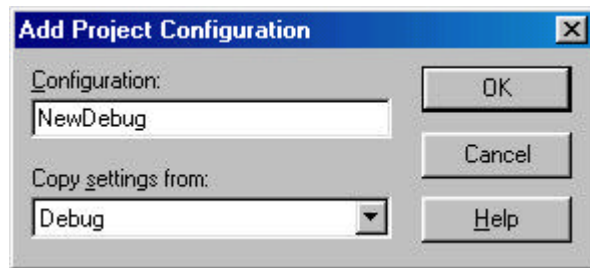
Setting up configurations

When 'Configurations' is selected from the 'Project' menu the following dialog box is displayed:



Each configuration is named and can then be changed as required. The Release and Debug configurations are default configurations may be added or removed at any time.

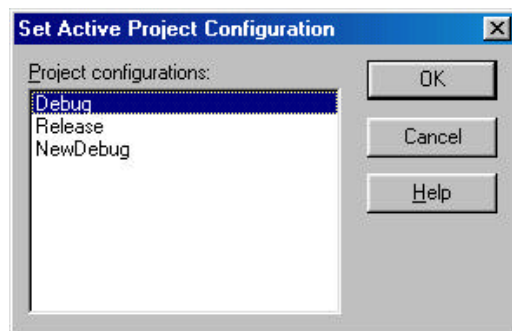
Selecting the Add option will cause the following dialog box to be displayed.



The configuration to be added must be given a new name. Initial settings for this new configuration must be taken from an existing configuration. The settings may then be changed as required.

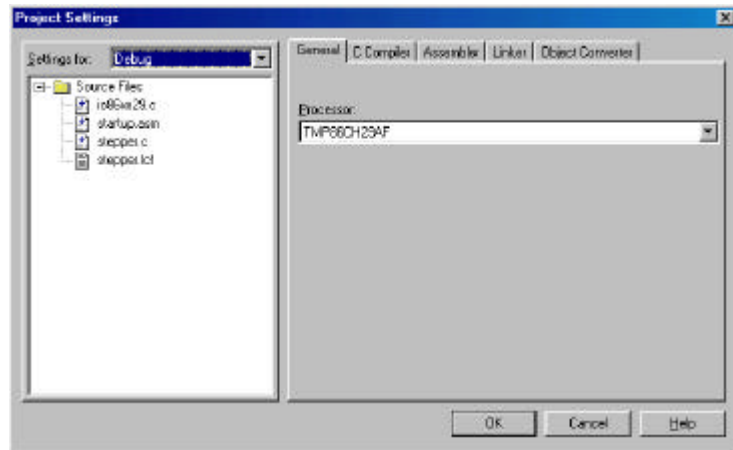
Selecting the active configuration

The settings and options defined for the active configuration will be those used during any Build or Rebuild action. To set the current active configuration select 'Set Active Configuration' from the 'Project' menu. The following dialog box will be displayed:



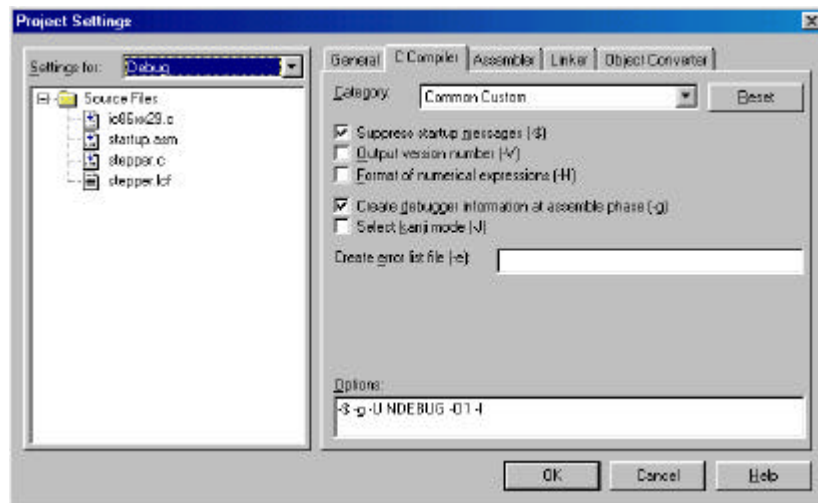
Changing the project settings

The change of the settings for a defined configuration select 'Settings' from the 'Project' menu. The following dialog box will be displayed:



The 'Processor' list box contains the processor type for which the project is being built. Note that this is not required for the rebuilding process but for creating the processor in the simulation. If the processor type is changed it is necessary to reload or rebuild the project before the change will take effect.

Options for the C Compiler, Assembler, Linker and Object converter are specified by clicking the appropriate button to display a dialog box from which the settings for the appropriate tool can be viewed and changed as required. As an example, when the C Compiler button is pressed a dialog box similar to the following is displayed:



The 'Setting for' list box indicates the configuration for which the settings are relevant. The list box will show by default the currently active configuration. The settings for other configurations may be changed by selecting the appropriate configuration from the list box.

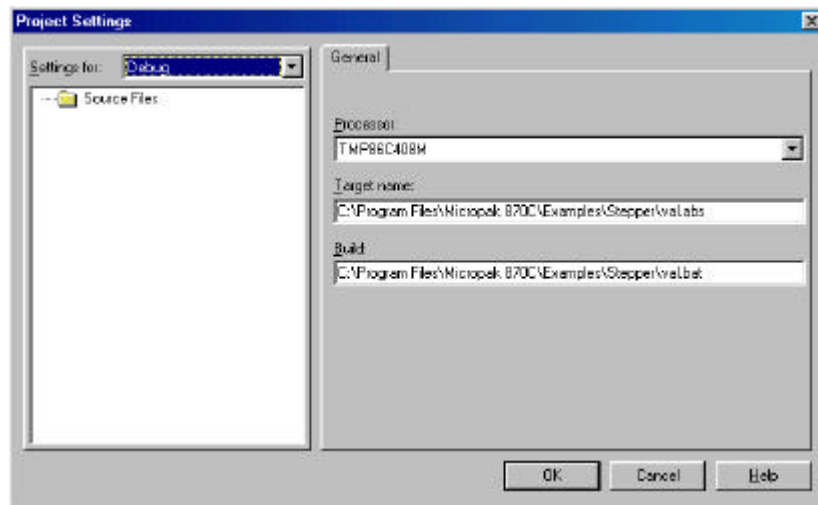
The 'Options String' is a non-editable text box which displays the options as they would appear if entered at the DOS command line.

The 'Category' list box displays the categories relevant to the selected build tool and the settings appropriate for the selected category are displayed in the 'Category Settings' window. From here you can specify the options required, with the changes being automatically reflected in the 'Options String' text box. Note that the options will only take effect when the 'OK' button is pressed.

The 'Reset' button will reset the options selected for the tab to the default settings.

External project options

When 'Project' is selected from the 'Options' menu (and the project is an external type) the following dialog box is displayed:



The 'Target Name' is the name of the debug or release executable file and is, by default, assumed to be the batch file name with a .ABS extension.

The 'Processor' list box allows you to select the type of processor for which the project is being built. This is not required for the rebuild process but is necessary for creating the processor for the simulation. If the processor type is changed a project reload or rebuild is required before the change will take effect.

The 'Settings for' list box allows you to specify which configuration you are specifying settings for,

Using the Editor

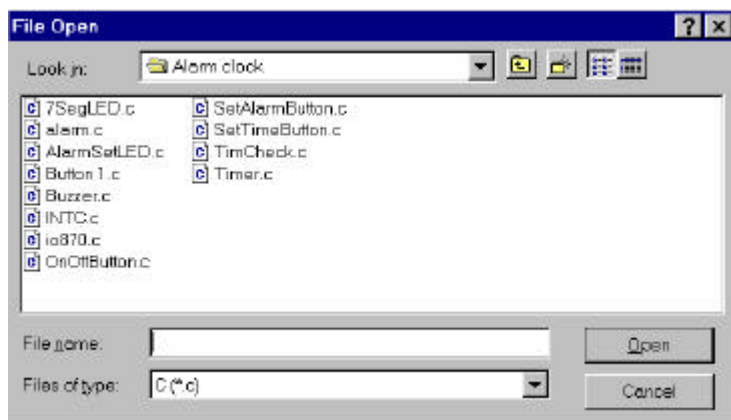
Micropak 870C allows text files to be displayed and edited.

Opening files

The normal method for opening a new file is by selecting 'Open' from the 'File' menu. 'Shortcut' methods are also available using the accelerator function keys or by clicking the following button on the tool bar:



The following dialog box is displayed:



Locate the desired file by selecting the required drive and directory and selecting from the list of files. Note that only files of a specific type will be displayed, according to the value selected from the 'List files of type' field, e.g.:

File specification	File type
*.c	'C' source files
*.cl, *.src	'C-Like' source files
*.mac	Macro processor files

*.app	Preprocessor files
*.asm	Assembler files
*.h, *.inc	Header files
*.lnk, *.lcf	Linker files
*.h16, *.h20	Intel hex files
*.s16, *.s24, *.s32	Motorola S files
*.cpl, .lst	Listing files
*.map	Map files
*.bat	Project files
*.scr	Script files
*.pan	Panel files
.	All files

File defaults

In addition to the above file types the following extensions are assumed defaults:

File extension	File type
*.bat	Project batch file
*.tst	Project context files (not text files)

Syntax colouring

As an aid to entering and colouring TLCS-870/C family source code a syntax colouring facility is provided. This colours the various elements of code as follows:

Element	Colour
Extended Keywords	red
Keywords	blue
Comments	green
Dis-assembled code	grey

Syntax colouring can be enabled by selecting 'Syntax Colouring' from the 'View' menu. Note that syntax colouring is only available for C source files.

Mouse driven functions when editing

The text cursor can be set anywhere simply by clicking with the left mouse button.

Selecting areas of text

Areas of text can be selected by dragging the text cursor from the beginning of the required area to the end of it. Text blocks selected in this way will be shown as white on black. Selected blocks of text are automatically copied to the 'find' string whenever the 'Find' or 'Replace' dialog boxes are opened.

The double-click

If the left mouse button is double-clicked whilst the text cursor is positioned within a word, it will be selected and editing will now be in word mode.

The triple-click

If the left mouse button is triple-clicked whilst the text cursor is positioned within a line, the text within that line will be selected and editing will change automatically to line mode.

The shifted left hand click

If the left mouse button is clicked whilst the Shift key is pressed, the text between the current text cursor position and the current mouse cursor position will be selected.

When the editor is in word or line mode, the selected area of text will include that previously selected by double-clicking or triple-clicking.

Editing possibilities

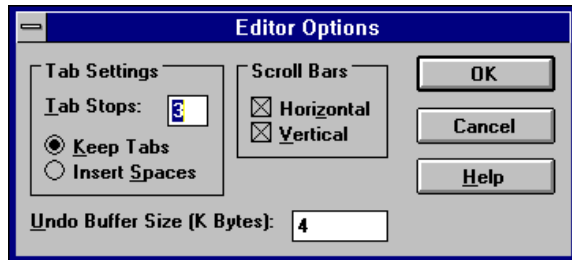
Once an area of text has been selected it can be:

- Deleted (Del)
- Removed from the file and placed in the clipboard (Cut)
- Copied to the clipboard (Copy)
- Pasted from the clipboard to another position (Paste)

These functions can be invoked via the 'Edit' menu or by using the appropriate keyboard accelerator shortcuts.

Editor options

Editor options are set by selecting 'Editor' from the 'Options' menu. The following dialog box is displayed:



The tab spacing can be changed to any number between 1 and 62. The default number is 4 characters.

The horizontal and vertical scroll bars can be deselected independently by clicking the appropriate check box, e.g.:



The 'Undo' buffer is used to store editor commands and associated text in order to enable actions to be subsequently undone by the 'Undo' command. The size of this buffer may be set between 0 Kbytes and 31 Kbytes.

Keyboard functions when editing

Basic text editing is performed by positioning the text cursor and using the standard editing keys and functions as shown below. To position the text cursor use the cursor keys or click with the left mouse button.

Key	Function
Ins	Toggle between insert and overwrite mode
Del	Delete character to right of cursor or previously selected block
Backspace	Delete character to left of cursor or entire block

Home	Skip to beginning of line
End	Skip to end of line
PgUp	Move one page up
PgDn	Move one page down
Left	Cursor one character left
Right	Cursor one character right
Up	Cursor one character up
Down	Cursor one character down
Ctrl+Left	Move one word left
Ctrl+Right	Move one word right
Ctrl+Up	Scroll window up one line
Ctrl+Down	Scroll window down one line
Ctrl+PgUp	Scroll left one page
Ctrl+PgDn	Scroll right one page
Ctrl+Home	Skip to start of file
Ctrl+End	Skip to end of file
Ctrl+A	Redo
Ctrl+Z	Undo
Ctrl+X, Shift+Del	Copy to clipboard and delete i.e. Cut
Ctrl+C, Ctrl+Ins	Copy to clipboard i.e. Copy
Ctrl+V, Shift+Ins	Copy from clipboard i.e. Paste
Ctrl+F	Find
Ctrl+R	Replace
Return	Insert new line

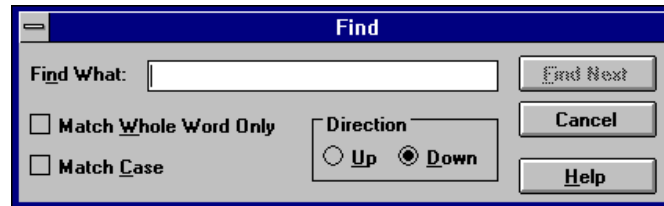
Locating and changing text

The following facilities are available for locating and, optionally, changing specific text within a file:

Facility	Function
Find	Locate a specified string of characters
Replace	Locate a specified string of characters and optionally replace it with a different string
Bookmarks	Set, clear and move between markers set in the text

Find

This is selected from the 'Edit' menu. The following dialog box is displayed:

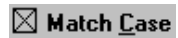


Type the string of characters to be located in the 'Find What:' field.

If the entire string is to be matched select the following check box:



If the case of the string of characters is to be matched select the following check box:

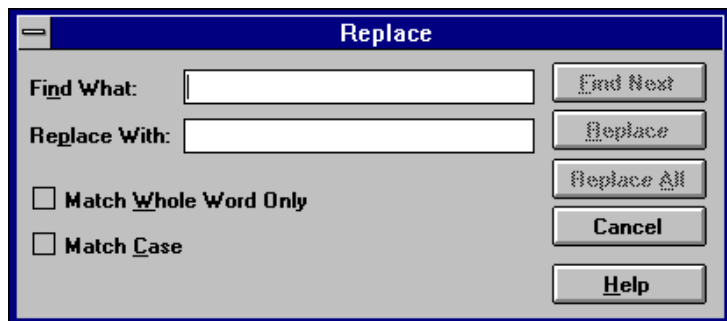


The direction of the search within the file can be specified using the following radio buttons:



Replace

This is selected from the 'Edit' menu. The following dialog box is displayed:



Type the string of characters to be located in the 'Find What:' field. Type the replacement string of characters in the 'Replace With:' field.

If the entire string is to be matched select the following check box:



If the case of the string of characters is to be matched select the following check box:



To locate next instance of the string to be replaced click the following button:



To replace the current occurrence of the character string and find the next occurrence click the following button:



To replace all occurrences of the character string click the following button:



Bookmarks

To set a bookmark, position the text cursor on the line to be marked and select 'Toggle Bookmark' from the 'View' menu. To subsequently remove a bookmark first locate it and then select the same menu option.

To move to the next bookmark in the file select 'Next Bookmark' from the 'View' menu.

To move to the previous bookmark in the file select 'Previous Bookmark' from the 'View' menu.

Note that these menu options have corresponding 'accelerator' key sequences.

To remove all bookmarks in the current file select 'Clear All Bookmarks' from the 'View' menu.

The implications of editing

When any relevant source areas are edited, the relationship between the object and source is altered and the object file no longer corresponds exactly to the source files. Therefore, the compiler or other relevant tool should be re-invoked to re-build the object file from the source file.

Re-building the project

The C compiler, or assembler can be re-invoked to re-build the object file from the source file either by selecting the 'Compile' option from the 'Project' menu. The 'Build' or 'Rebuild All' options may also be selected from the 'Project' menu to cause a complete project build including the link and object converting processes. Tools are available on the tool bar for the build commands. Selecting 'Build' will cause only those files which have changed to be re-compiled.

When the tool is re-invoked the re-build process will run and its output will be displayed in the output window. This will include any errors encountered by the tool including compiler, assembler, linker and converter errors.

When the process is complete, the output window should be examined for errors before continuing. On closing the output window, Micropak 870C will then read the new source and object files and redraw the display.

Please note that the re-building process will operate on the disk stored versions of the source files. If the editor has been used and the file not stored a message will be displayed asking if you wish the file to be stored. You may choose to store the file or to continue the rebuild with the old file.

The re-build mechanism

This mechanism relies on the project batch file and a number of related files being set up correctly, which is discussed in more detail in the 'Navigating Project Files' chapter of this user manual.

Correcting flagged errors

Any errors found are flagged. The 'View' menu lists 'Next Error' and 'Previous Error' menu items. Selecting these items enables you to locate each error.

Double-clicking on an error in the output window will also take you to the appropriate line in the source file.

Controlling Execution

Overview

There are various ways in which the TLCS-870/C code loaded into the simulated system can be executed, such as single step, multi-step and so on. Before describing each of these in detail let us consider some general aspects of execution, such as how to stop the processor when it is running and the factors which affect execution speed and how they can be optimised.

Ceasing execution

Execution can always be interrupted by pressing the appropriate function key, selecting 'Stop Debugging' from the 'Debug' menu or clicking the following tool bar icon:



Optimising execution speed

Updating view items on-screen slows the simulation, as do complex breakpoints. To get the optimum execution speed you should restrict the view items that are visible on-screen and reduce the number of active breakpoints to a minimum.

The first of these options can be achieved by 'minimising' any windows which are not currently being monitored by clicking the respective minimise button:



Note that source windows may also be minimised.

Breakpoints can be reduced by disabling those which are not required for the current test.

Execution possibilities

The code can be run in Micropak 870C in the following way:

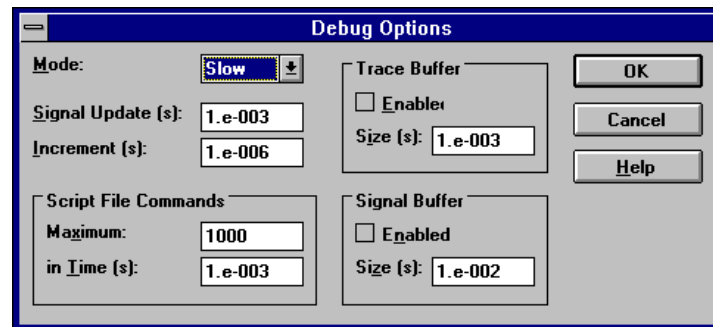
- Go (i.e. run until breakpoint is reached)
- Step into
- Step over
- Step to cursor
- Step out

Each of the above has a corresponding icon on the tool bar:



Simple execution and debugging modes

This option allows execution to continue until a breakpoint is reached, or until execution is halted. Two modes of operation are possible: 'Slow' and 'Fast'. The mode is specified by selecting 'Debug' from the 'Options' menu. The following dialog box is displayed:



Select from 'Slow' or 'Fast' in the 'Mode' field of this dialog box.

'Slow' mode will show you multi-step animation of the code, updating all windows and showing the progression of the PC through the active source window after each instruction is executed. 'Fast' mode will only update the test panels and the animated PC will not be shown.

The modes of operation may also be specified by clicking the mode icon on the tool bar, which toggles between 'Fast' and 'Slow' modes:

'Fast' mode: 'Slow' mode:



Single stepping - Step into

Clicking the 'Single Step' icon or pressing the appropriate function key will cause the execution of the target code pointed to by the current PC.

Single stepping - Step over

This option, similar to 'Step Into', will execute one instruction or all instructions within a single procedure, if the statement pointed to by the PC is a call to a procedure.

Step out

This will run to the first RET instruction which occurs with the stack at the current level. This is provided as a fast way of running to the end of a subroutine once the code in an area under investigation has been stepped through. The stack level condition implies it will ignore any function/return combinations encountered on route.

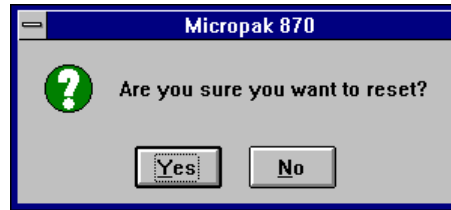
Step to cursor

This allows the code to run until execution reaches the line in the source window where the cursor is placed.

To use this function, first set the cursor by clicking within the desired line. The function can then be invoked by pressing the appropriate function key or clicking the 'Step Cursor' icon on the toolbar.

Resetting and viewing the processor clock

The processor may be reset by selecting 'Reset' from the 'Debug' menu. The following confirmation box will be displayed to ensure that the processor is not reset accidentally:



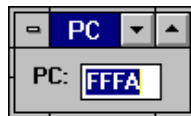
A reset generated in this way will set the processor back to a clock value of '0' and all the normal processor actions expected at reset will occur.

The processor clock may be viewed in a device window by selecting 'Device' from the 'Window' menu and choosing from the list.

The program counter

Micropak 870C keeps control of execution through the program counter (PC). This is updated whenever a statement is run.

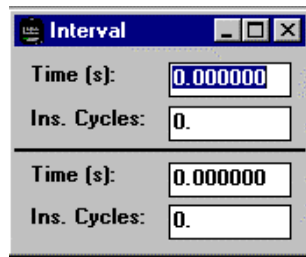
The current position of the PC is seen in the source window as a yellow bar. It can also be viewed by selecting 'Device' from the 'Window' menu and clicking the 'Program Counter' option. The following window will be displayed:



The PC can be set using this dialog box or by choosing 'Set PC to Cursor' from the 'Debug' menu, which sets the PC to the ROM address currently being pointed to by the cursor.

The interval window

An interval window can be selected for display by selecting 'Interval' from the 'Window' menu. The following window is displayed:

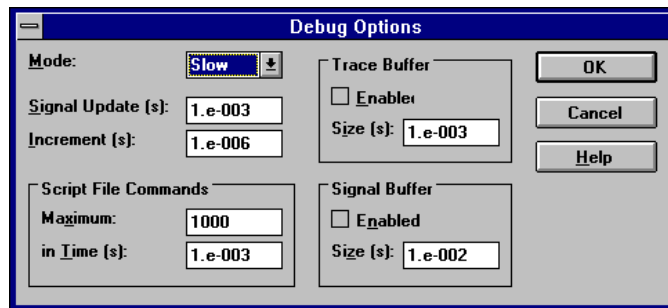


The interval values will be incremented with each successive execution.

The interval value may be reset by selecting 'Clear Interval' from the 'Trace' menu.

Debug options

A number of user configurable options are available by choosing 'Debug' from the 'Options' menu. The following dialog box will be displayed:



Mode

Fast

This run mode allows simulation execution at full speed. Only the test panels are updated.

Slow

When this run mode is selected all relevant windows will be updated after each instruction is executed.

Signal window control

Signal Update(s)

A value of simulated time in seconds may be input. This value specifies the time interval between each signal window update.

Increment(s) Again, a value of simulated time in seconds may be input. This value will be used when there is no activity in the system and the CPU is in either HALT or STOP mode. In this instance the value will be used to increment the clock and thus accelerate the simulation.

Script file commands

This facility is provided to allow the simulator to trap script file loops. Two parameters are used:

Maximum Here the user specifies the maximum number of script instructions to be executed in a given time.

in-Time(s) The user specifies the time limit.

If the number of commands executed within the time specified reaches the maximum specified a warning is given.

Trace buffer control

Enabled Check box for enabling or disabling trace buffering.

Size(s) The user can specify the size of the trace buffer which is expressed as simulated time, in seconds.

Signal buffer control

Enabled Check box for enabling or disabling signal capture.

Size(s) The user can specify the size of the signal buffer which is expressed as simulated time, in seconds.

Trace Options

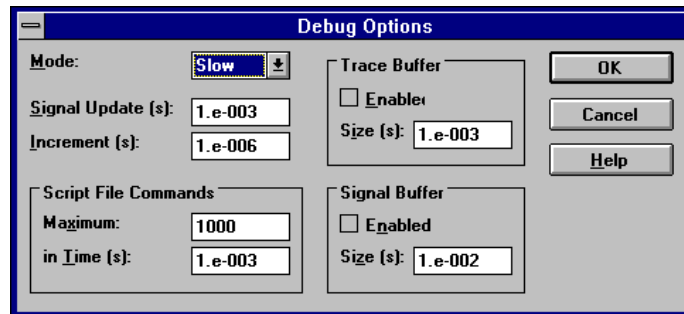
Trace Buffering

Micropak 870C includes a trace buffer. This records the significant aspects of the target system after each instruction has been executed. When execution ceases (such as when a breakpoint has been encountered) it can be used to show how control reached the current point.

Note that operating with trace buffer active requires Micropak 870C to store a significant volume of data after each instruction and consequently slows execution speed considerably.

Controlling Tracing

Tracing can be activated by selecting 'Debug' from the 'Options' menu. The following dialog box will be displayed:



Click the 'Enabled' check box to activate the trace facility:



You can also specify the depth of the trace buffer in seconds by typing a value into the 'Size' field.

Trace buffer displays

Once a set of execution history information has been captured in the trace buffer, Micropak 870C allows you to 'roll-back' the displays in order to show the information captured. This effectively reverses the direction of execution through the normal listing display.

For example, in this 'roll-back' display, single stepping causes the previously executed line of code to be the active line rather than the following line. The active line is shown in green.

Invoking roll-back displays

The following roll-back displays are available from the 'Trace' menu:

- Go back (i.e. back to start of last execution)
- Step back into
- Step back over
- Step out
- Step back to cursor

Rolling back is limited by trace buffer size

Because 'roll-back' displays operate by interpreting the path through the code recorded in the trace buffer, the display can not be 'rolled back' to instructions executed earlier than the oldest recorded record in the trace buffer. Once the beginning of the buffer is reached a message will be displayed.

In practice the size of the trace buffer is limited and this implies a limit on how far the trace buffer can be 'rolled back'.

The depth of the trace buffer is specified on the 'Debug Options' dialog box, accessible from the 'Options' menu.

Stepping forward through the buffer

When execution has been rolled back, it is then possible to step forward again through the buffer until the end of the buffer is reached.

Any one of the options on the 'Debug' menu for controlling execution may be used for this task. The PC bar will continue to be shown in green whilst the buffer is being traversed. Once the end of the buffer has been reached a message will be displayed and the PC bar will revert to yellow.

Restarting execution

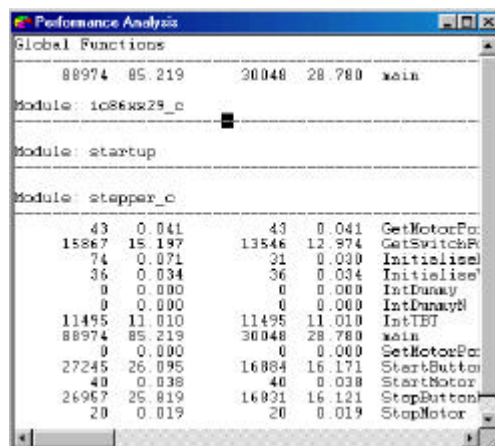
In order to restart real execution the trace buffer display must be at the end position. If the trace buffer is not at the end position you should issue a 'Go' command to step through the buffer to the end and then instigate the required execution.

Inactive trace buffer

If the trace buffer is inactive, no roll back will be possible. Under these conditions the menu and tool bar items associated with rolling back through the trace buffer information will be 'greyed out'.

Performance Analysis

Whenever Micropak is executing a trace is kept of the path of execution through function calls. This trace can be displayed at any time by selecting 'Performance Analysis' from the 'Window' menu.



The screenshot shows a window titled 'Performance Analysis' with a list of function calls and their execution times. The window is divided into sections for 'Global Functions', 'Module: 1c86xx29_c', 'Module: startup', and 'Module: stepper_c'.

Address	Time	Address	Time	Function Name
88974	85.219	30048	28.780	main
Module: 1c86xx29_c				
Module: startup				
Module: stepper_c				
43	0.041	43	0.041	GetMotorPo
15867	15.197	13546	12.874	GetSwitchP
74	0.071	31	0.030	Initialize
36	0.034	36	0.034	Initialize
0	0.000	0	0.000	IntDummy
0	0.000	0	0.000	IntDummy
11495	11.010	11495	11.010	IntTET
88974	85.219	30048	28.780	main
0	0.000	0	0.000	SetMotorPo
27245	26.095	16884	16.171	StartButton
40	0.038	40	0.038	StartMotor
26967	25.819	16931	16.121	StopButton
20	0.019	20	0.019	StopMotor

Performance Analysis Data

Each function is listed within the code coverage window and for each function two sets of figures are listed. Each set shows an execution time and the percentage of this time against the total execution time.

The leftmost set gives the data for the function named including all dependent functions. The rightmost set gives the data execution within the function named alone.

Clearing Performance Analysis Data

The data is automatically cleared whenever a processor reset is encountered. The data will not be cleared between different execution requests.

You should select 'Clear Performance Analysis' from the Trace menu in order to force the data to zero.

Code Coverage

Executed code may be viewed in the source window. All code executed is coloured grey. The execution coverage area can therefore be seen. To use this the Code Coverage option must be activated by selecting 'Code Coverage' from the 'Trace' menu. Selecting the option again will deselect Code Coverage.

Clearing Code Coverage

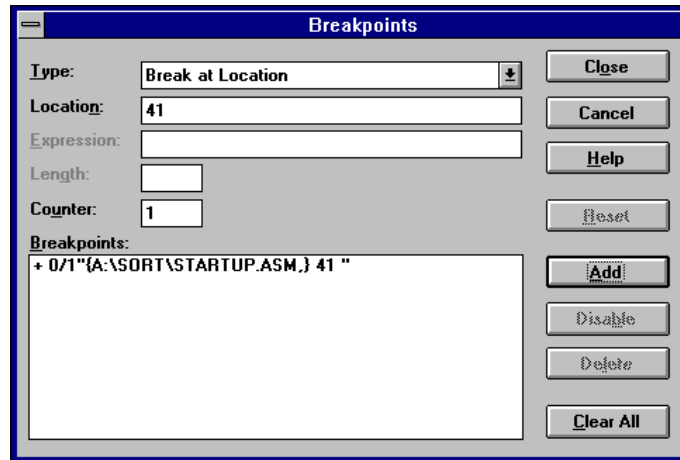
The data is automatically cleared whenever a processor reset is encountered. The data will not be cleared between different execution requests.

You should select 'Clear Code Coverage' from the Trace menu in order to force the data to zero.

Breakpoints

Setting breakpoints

Selecting the 'Breakpoints' option from the 'Debug' menu will cause the following dialog box to be displayed:



This dialog box is used to add, remove, enable or disable breakpoints.

Breakpoints in Micropak 870C are based on the idea of the value of an item (location or 'variable') in the simulated system matching, according to a known relationship, a nominated value. This can be thought of as BOOLEAN expression - if the expression evaluates to true the breakpoint condition has occurred, and when it evaluates to false it has not.

The fields in the upper part of the dialog box aid the construction of suitable breakpoint definitions, whilst the lower section of the window shows a list of currently defined breakpoints in a shorthand form.

The example shown in the dialog box is of a breakpoint set on a source instruction to be met after one occurrence. In more detail the elements are:

Element	Explanation
+	Breakpoint enabled (disabled breakpoints are shown with '-'). This is controlled using a dialog box button.
0/1	Breakpoint will be reached after one occurrence. This is input by the user in the 'Counter' box.
A:\sort\startup.asm	The breakpoint is set on a location in the given source file.
41	<p>The breakpoint is set to monitor ROM locations. The ROM locations are specified as those relating to a given line in a given source file</p> <p>The 'Type' box will show a selection list of possible breakpoint monitor types and the required type is selected by the user.</p>

Further details of the possibilities allowed in each box are given in the following sections.

The 'Type' field

This field allows the selection of a number of breakpoint types, these being:

Break at location

Micropak 870C allows execution breakpoints to be set on a memory location specified in the 'Location' edit box.

Break at location if expression is true

An expression specified in the 'Expression' edit box is checked when the PC reaches the location specified in the 'Location' edit box. If the expression evaluates to a non-zero value then the breakpoint is considered to have been reached.

Break at location if l-value has changed

An l-value, or data item may be specified in the 'Expression' edit box. This data item is checked when the PC reaches the location specified in the 'Location' edit box. If the data item has changed in any way since the last execute command was given then the breakpoint is considered to have been reached.

Break when expression is true

If the expression specified in the 'Expression' edit box evaluates to a non-zero value then the is considered to have been reached.

Break when l-value has changed

If the l-value (data item) specified in the 'Expression' edit box has changed in any way since the last execute command was given then the breakpoint is considered to have been reached.

The 'Location' field

This edit box is only enabled for the 'Break at location', 'Break at location if expression is true' and 'Break at location if l-value has changed'. Various formats for this edit box are permitted. A number indicates an absolute ROM address and can be specified in either decimal, hexadecimal (preceded by '0x' or '0X') or octal (preceded by '0'). The alphabetic digits for hexadecimal numbers can be in either lower or upper case. If a number is prefixed with a period ('.') then it is assumed to be the line number of the active source file. Other files can be referenced by prefixing the line number with the filename and a pling ('!').

The 'Expression/L-Value' field

This edit box is enabled for all breakpoint types except the 'Break at location' type. The text associated with it changes depending upon the breakpoint type selected. If a 'Break at location if expression true' or 'Break when expression is true' breakpoint is selected then the text is 'Expression'. If a 'Break at location if l-value has changed' or a 'Break when l-value has changed' breakpoint is selected then the text is a data item. Only static data items are allowed and they must be entered by symbolic name.

The edit box accepts a limited C syntax for specifying expressions as follows:-

Item	value or operator
Identifier	Any static variable name allowed

Constant	Any integer value
And expression	&
Equality expression	== ,!=
Relational expression	<.> <=,>=
Unary expression	-

The 'Length' field

This edit box is enabled for the 'Break at location if l-value has changed' and 'Break when l-value has changed' breakpoints. It gives the number of objects defined in l-value which are to be interrogated for changes. The default length is '1' and this should only normally be increased if the l-value makes reference to an array of objects.

The 'Counter' field

This edit box is used to specify the number of times that the breakpoint is to be met before it is considered to have been 'hit'. The default value is '1'.

Adding a breakpoint

The breakpoint details must be specified by filling out the relevant edit boxes as described in the previous sections. Once this has been completed the 'Add' button should be selected. The breakpoint details will then be transferred to the lower section of the 'Breakpoints' dialog box.

Viewing current breakpoints set ups

The 'Breakpoints' dialog box shows a list of current breakpoints and their status. This list is scrollable and one can see all breakpoints set. Clicking on any one of the breakpoints displayed will transfer all the details to the fields in the upper part of the box enabling the detail of the breakpoint to be easily checked.

Setting breakpoints in the source window

Execution breakpoints on a single address may be set directly when displaying the source window. This may be achieved by moving the cursor to the required line in the window and clicking the 'Toggle Breakpoint' icon to set the breakpoint.



Set a breakpoint by using this icon

The source window will show the line on which the breakpoint is set in red. Note that the F10 key may also be used for this function.

Removing breakpoints

To remove a breakpoint, the dialog box should be activated and the breakpoint to be removed selected from the list. On selection the details of the breakpoint will be transferred to the individual boxes so that the user can verify his selection. The 'Delete' button in the dialog box should then be clicked and the breakpoint details will be removed from the list.

For single location breakpoints which can be seen in red in the listing window, the 'Toggle Breakpoint' icon can be clicked. If the cursor is positioned on the breakpoint and the button clicked the breakpoint will be removed.

Enabling/disabling breakpoints

Individual breakpoints may be temporarily disabled and then re-enabled at any time using the dialog box. The required breakpoint should be selected and the 'Disable/Enable' button clicked accordingly. The current status of each breakpoint is denoted by '+' (enabled) or '-' (disabled) in the breakpoint list box.

Script file facilities

Actions in script files can be controlled and triggered either by event triggers or by script file breakpoints.

Breakpoints may be set to correspond with breakpoints set within the simulation. The script language contains a 'breakpoint' function which allows the user to specify each of the breakpoint conditions described above.

Further event triggers can be specified through the script language 'add' command. Here, triggers can be set to cause an event on:

- Test panel input
- Port activity
- Time-out

When a script file breakpoint or event trigger is reached the action routine referred to will be triggered and execution of the target code suspended until the actions have been processed.

Port Simulation Techniques

Overview

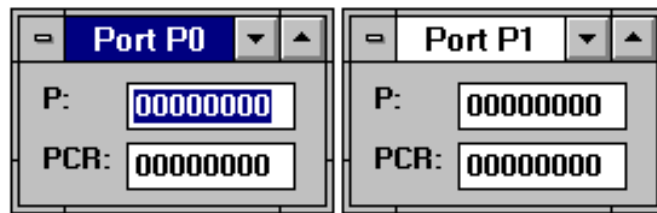
The simulation includes, for each port line, an internal and an external ‘Thevenin’ equivalent network, each of which consists of a single voltage generator and a single series resistance. The internal network is provided and controlled by the simulation in accordance with the internally set port controlling conditions such as the state of the port data registers, etc.

For the external network both the voltage and series resistance can be controlled via the user interface or the script file.

Extending the simulation in this way allows the interaction of the firmware under test with external hardware elements such as switches, LEDs, etc., to be investigated, allows checks on the drive capabilities of the ports and the use of pull-ups, etc.

Pin and port windows

The Micropak 870C simulation includes all the significant pins of the target processor. It is possible to invoke a window for any additional pin. In addition, it is possible to show summaries of the pin information grouped as ports. These port views show the logical values at the port.



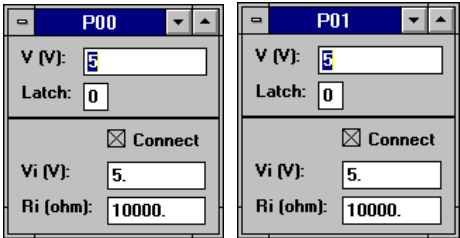
Grouped pin information for a port

Both pin and port views can be updated by overtyping the displayed fields.

An external voltage/resistance network for every pin

An external voltage/resistance network is simulated for every pin. The external ‘Thevenin’ network is automatically controlled by the simulation and consists of a single resistance and a single voltage source.

The voltage and the resistance values are specified by the user and can be changed if desired whilst the target code is running. Furthermore, the network may also be connected or disconnected from the pin by toggling the ‘connect’ box in the pin window.



A network is provided for each pin

Assumed port characteristics

The assumed characteristics of the port hardware are as follows:

Item	Assumed value
Pull-up value	70k ohms
Top CMOS driver	570 ohms
Bottom CMOS driver	250 ohms
Bottom high current CMOS driver	50 ohms

Using these networks

Including a simulation of these simple internal and external networks for each pin allows checks on the behaviour of the internal port hardware.

Consider, as an example, a case in which a particular port line was intended in a design to be used as a permanent input, sensing the value being fed to it by a standard CMOS buffer. In this case it would be sensible to mimic the external CMOS buffer driver by setting the resistance value to the output impedance of the buffer (say 100 ohms) and then specifying the data conditions into the pin by setting the voltage of the voltage generator to VCC or 0 (for logic '1' or '0' respectively).

Using the simulated network in this way allows the firmware under test to check the programming of the port. Assuming that the port line was correctly programmed as an input, the voltage at the port input would follow closely the rail to rail changes made in the voltage generator setting. However, if there was a bug in the firmware under test, such that the internal NMOS buffer on this line was inadvertently activated, the voltage shown for the pin would deviate significantly from the expected values, highlighting the drive contention and drawing attention to the bug.

Using script files to control port conditions

Script files can be used to set port conditions using the statements:

connect, setr and setv

Each of these statements are described below:

connect <pin>,<on/off>

This connects or disconnects the external network, depending on the single parameter given (0 = disconnect, non-zero = connect). Disconnection is similar in effect to setting the series resistance to an infinite value.

setr <pin>,<resistance>

The setr statement takes two parameters, the first defining the pin and the second the series resistance directly in ohms.

setv <pin>,<volts>

This statement controls the external voltage generator associated with the pin, using a parameter which defines the desired value directly in volts.

Here is an example sequence of statements which would connect an external voltage source of 3.6 volts to pin port 2.1 via a series resistance of 47 k ohms:

```
setv 19,3.6 :      rem 3.6 volts  
setr 19,47000 :    rem via a 47 k R  
connect 19,1 :     rem connect the external network
```

Note that the script file facilities provide a flexible mechanism which allows the parameters of the network to be changed intelligently to provide, for example, sinusoidal input voltages or switched loads or pull-ups.

Using script files to check port conditions

The current voltage at any pin is available to script file programs via the `getv` expression:

```
getv(<pin>)
```

This takes the port pin number as a single parameter and returns the pin voltage in volts.

Thus, for example, if the pin were to be used to drive a CMOS gate, which was regarded as having a fixed login threshold of 2.1 volts, the script file could derive the effective logic value as follows:

```
let CMOS_VALUE=(getv(19) >= 2.1)
```

For more details on using script files, see the associated section.

Pin numbering

The pin numbers used in these script statements correspond to the real pin numbers on the standard package for the simulated device.

Viewing Simulated Objects

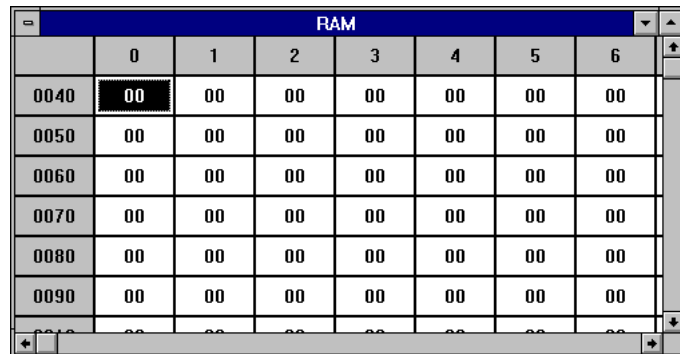
Overview

Micropak 870C provides a number of device windows which allows the states and activities of simulated objects to be viewed and monitored. These windows are updated whilst the simulation is running in slow mode or at the end of execution if running in fast mode. Micropak 870C also provides 'Test Panel' facilities to enable users to customise the way in which they wish to view items.

Displaying RAM

The RAM window

The basic memory view facility is composed of an array of values held in a RAM window. It is displayed by selecting 'Device' from the 'Window' menu then choosing 'Random Access Memory' from the list. The following window will be displayed:



	0	1	2	3	4	5	6
0040	00	00	00	00	00	00	00
0050	00	00	00	00	00	00	00
0060	00	00	00	00	00	00	00
0070	00	00	00	00	00	00	00
0080	00	00	00	00	00	00	00
0090	00	00	00	00	00	00	00

The window may be sized to include the RAM addresses required. Where the RAM addresses are not consecutive, this can be accommodated by opening multiple RAM windows.

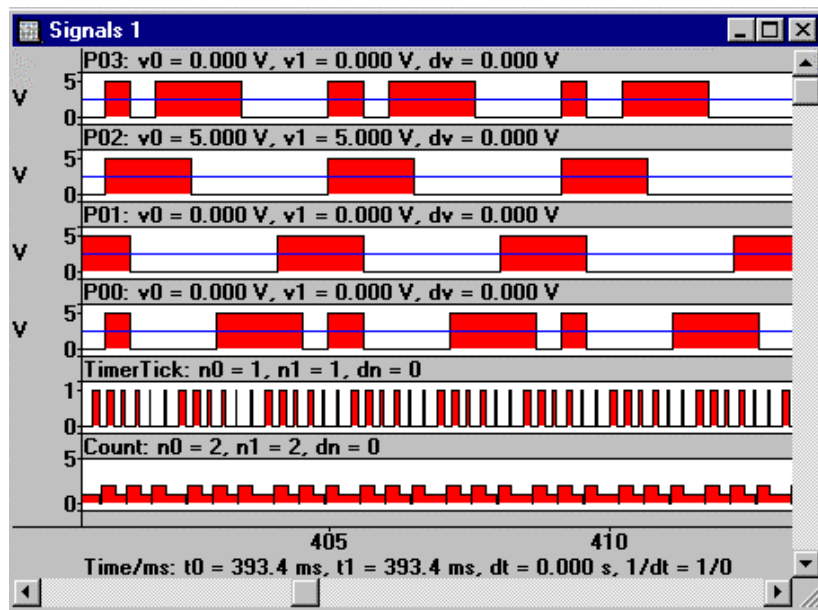
Customised memory views

Test panel windows can be used to show customised views of memory locations. These panels provide a good ‘application’ view of memory, as they allow location data to be output in a form specified by the user as applicable for the task. RAM may be displayed directly or may, for example, be shown translated into text as decimal numbers to mimic an application display. Further information about test panels is given later in this section.

Signal recording boxes

What is a signal recording box?

Micropak 870C allows display of pin values as time-based plots. These are shown in ‘signal recording boxes’ and resemble oscilloscope traces. Here is an example:

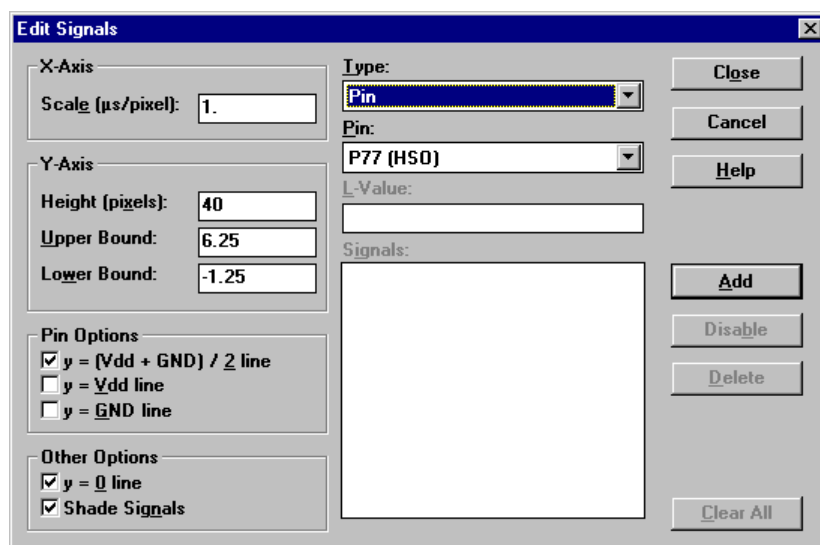


Example signals box

Setting up a signal box

To set up a signal box you must first select 'Signals' from the 'Window' menu. An empty signal box window will be displayed. Items to be plotted in the window can then be added by selecting 'Edit Signals' from the 'Trace' menu.

The following dialog box is displayed:

**Type**

Signals can be plotted for pins or data memory. The selection can be made through the type box.

Pin selection

The pins to be included within any one signal window can be selected from a list of valid pins. This list will be activated only if the pin type has been selected.

Memory selection

Data items or bits may be selected from the memory type box. Symbolic names may be given for data locations. These may be chosen from a selection list. A module name selection list allows the user to specify the module in which relevant symbols are declared. Alternatively an absolute address may be given. Bit addresses should be given with a period separating the address and the bit number e.g. 70.1. If a length is specified for a data item the value plotted will then be taken from the whole memory range.

The signal list

Once an item selection has been made, clicking the 'Add' button will cause the item to be listed in the 'Signals' list. One plot line will be included within the window for every item shown in the 'Signals' list.

Axes and scales

The scale of both the X and Y axes for the signal plot can be set by the user. The X axis is scaled in pixels per second and the Y axis in pixels per volt. Minimum and maximum values can also be specified for the Y axis, in volts.

Pin Options

Three check boxes are provided so that the user can specify marker lines they wish to be included in the signal window for pin plots. The following options are available:

(Vdd + GND)/2	This is set at the cross-over point between logical values 0 and 1.
GND	This is set at the ground voltage.
Vdd	This is set at the power supply voltage.

Other Options

Two check boxes are provided for users to specify options they wish to include on pin and memory signals. These are as follows:-

0 line	A marker line is set at zero volts.
Shade signals	Shading between the baseline and the plot line is included

Removing plot lines

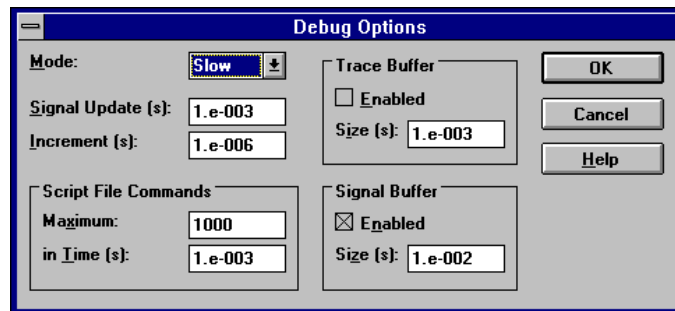
Individual plot lines can be removed from the window by selecting the signal from the 'Signals' list and then clicking the 'Delete' button. To remove all plot lines click the 'Clear All' button.

Enabling/disabling plot lines

Individual plot lines can be temporarily disabled and then re-enabled. This is achieved by selecting the plot line from the 'Signals' list then clicking the 'Enable' or 'Disable' buttons as appropriate.

The signal buffer must be enabled

The signal window will only be updated with information once the target code has been executed, so initially the window will be devoid of signal information. Furthermore, as updating the signal window slows the execution speed, a check box is provided on the 'Debug Options' window (displayed by selecting 'Debug' from the 'Options' menu) allowing you to enable/disable the signal buffer:



Note that this dialog box also allows you to specify the size of the signal buffer which can hold historical information about the plots and the frequency of the window updating. Further details on this are given in the section on Debug options.

Viewing the results - zooming and snapping

Once the signal information has been generated the window will be updated with the results. The window is scrollable so that the historical information held in the signal buffer can be viewed.

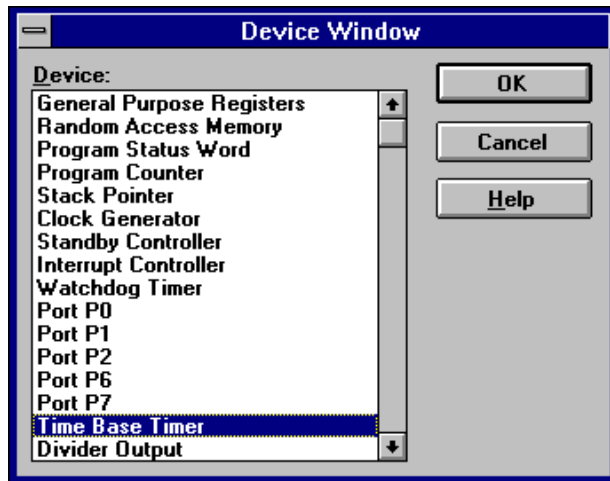
You can also use the ‘Signal Zoom In’ option from the ‘Trace’ menu to obtain a more detailed look at any specific part of the signal generated. A ‘Signal Zoom Out’ menu option is also provided if only an overview is required.

Each time ‘Signal Zoom In’ or ‘Signal Zoom Out’ are selected the scale of the display is changed by a factor of two. The maximum and minimum scales are one million pixels per second and one pixel per second. The size of the signal buffer is not linked to this scale. If a scale is selected in which part of the display exceeds the size of the buffer, a blank signal will result.

If you choose ‘Snap Signals’, the signal window display will show the nearest point at which the value of the item changed. Note that this may be a position either forward or backward in the signal buffer.

On-chip peripherals

All significant on-chip peripherals have associated view boxes. These can be viewed by selecting ‘Device’ from the ‘Window’ which displays a list of all valid devices for the chosen processor.



Double-clicking a entry in the list displays the selected device window, e.g.:



The device window shows, in information fields, all state and numerical information relevant to the device. The fields within the device window can be overtyped or changed using selection list options if specific conditions are required. If necessary, this can be done whilst the simulation is running.

Device Information

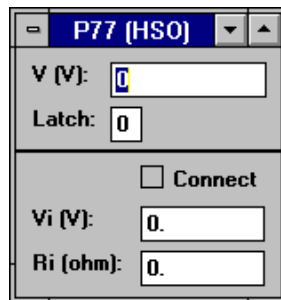
The types of devices available and the details associated with each are processor dependent. Therefore, the data sheets for the particular processor being used should be consulted for the relevant information about the devices and their constituent parts. Toshiba provide this information on CD and this data can be accessed directly from Micropak 870C using the 'Device Information' option from the 'Help' menu. To select device information, the focus must be set on the appropriate device window.

Port views

Port lines may be viewed individually as pins or in summary as a port window.

A port line is selected by double-clicking from the list displayed when the 'Pin' option is chosen from the 'Window' menu. The pin window shows the effective voltage on the pin and allows the specification of an external network to be connected to the pin.

To display a complete port, select 'Device' from the 'Window' menu then double-click the required device from the list, e.g.:



Port views give the pin values of each individual pin.

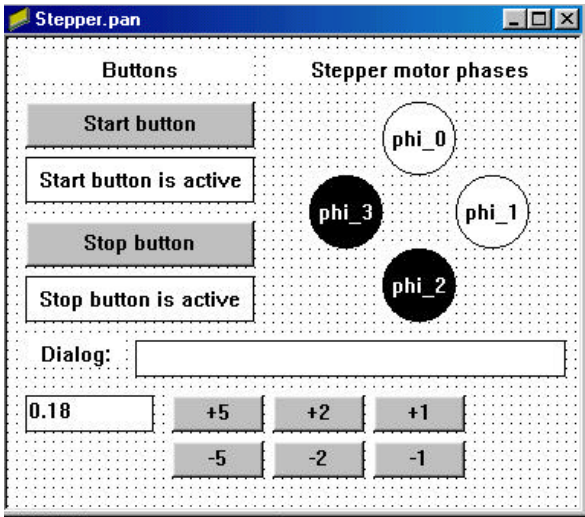
The chapter on 'Port simulation techniques' gives further information.

Test panel displays

What are test panels?

Test panel displays allow you to draw together a set of items, all related to a particular test set-up, to form a display which is particularly convenient for the tests to be carried out.

The display can be set up to show the states and contents of items such as port conditions, memory states and script file variables. It provides users with a convenient 'front panel' through which tests can be controlled and their results monitored. An example is shown below:



Test panels allow compact views to be built up which keep related objects together in the display. They allow, for example, the states of the port lines associated with the keys in a key scanning routine to be displayed in a format which can accept key presses.

The resulting input facilities and display can be particularly clear and convenient when debugging, for example, the key scanning module of an application.

Test panel options

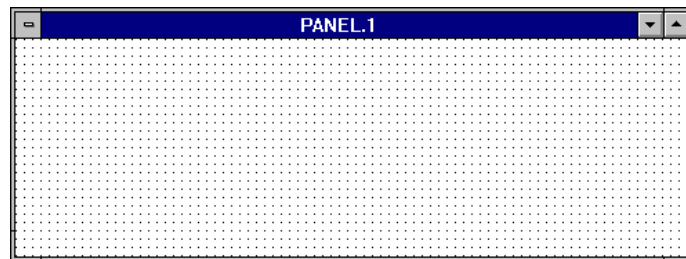
Options available from the 'File' menu and 'Test' menu allow test panels to be created and edited.

Menu	Panel Option
File	New Panel. Select this option to create a new panel.
File	Open. Select this option to open an existing panel.
Test	Show Panel Palette. This option is available when a panel is open and displays the palette window.
Test	Panel Properties. Select this option to specify or change the properties associated with an item.
Test	Panel Grid Settings. Use this option to change the grid settings for the panel and turn 'snap to grid' on or off.

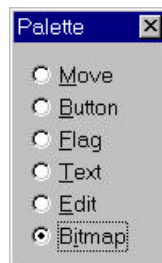
Items in each panel can be added to the panel and subsequently moved, edited or deleted. Each item has a set of properties associated with it (e.g. a caption to describe the item). These properties can be changed by selecting the 'Panel Properties' option from the 'Test' menu. To aid the user when aligning items in panels a grid is displayed. The 'Grid Settings' dialog box, displayed when the 'Panel Grid Settings' option is chosen from the 'Test' menu, can be used to turn the grid on or off.

Setting up a test panel

To set up a test panel select 'New Panel' from the 'File' menu. An empty panel window will appear, e.g.:



Items can then be added to the panel using the palette:



The tasks allowed using the palette are described below:

Move

This option allows defined items to be moved in the test panel using the mouse. Also, if a double-click is performed when the mouse is positioned on an item, the 'Properties' dialog box for that item will be displayed.

Flag

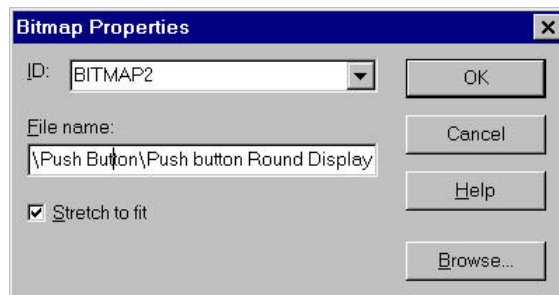
The 'Flag' option specifies a test panel flag item which can take one of two values ('0' or '1'). With default properties, the shape of the flag will be rectangular and the condition of the flag will be shown in inverse video (i.e. active) if it has the value '1'. The display of the flag can be customised with properties options so that:-

- Colours may be selected to both for background and for the caption text.
- Different colours may be chosen to differentiate between active and inactive states. to be shown in different colours when active and inactive.
- Different shapes may be selected for each flag.

The state of the flag is controlled by events in the script file and can, for example, be set to represent the state of a port line.

Bitmap

This option operates in a similar way to the 'Flag' option however the item displayed is a bitmap image file rather than a simple shape. With default properties set a simple rectangle is shown. This can be replaced by an actual bitmap image using the test panel properties or from specific script file instruction.



The bitmap file name must be entered together with the option the fit the bitmap within the area marked on the test panel.

In this way bitmap images may be used for example to represent the two positions of a toggle switch. The switch may initially be set up to display the off bit map and the script file mechanism may then be used at the appropriate time to change the image to the on position.

Button

The 'Button' option allows button event inputs to be entered by the user. The buttons are displayed in normal video and the mouse may be used to click on the button. The displayed button will then be set to show its 'pushed' state. When the button is released, an event associated with the button press will then be generated. This mechanism allows users to control events in script files. For example, a button event may be used to trigger a script file event which will set the conditions required for a simulated real button press in the hardware.

Text box

This option allows text to be displayed in a test panel. This enables the user to annotate test panels, making testing easier.

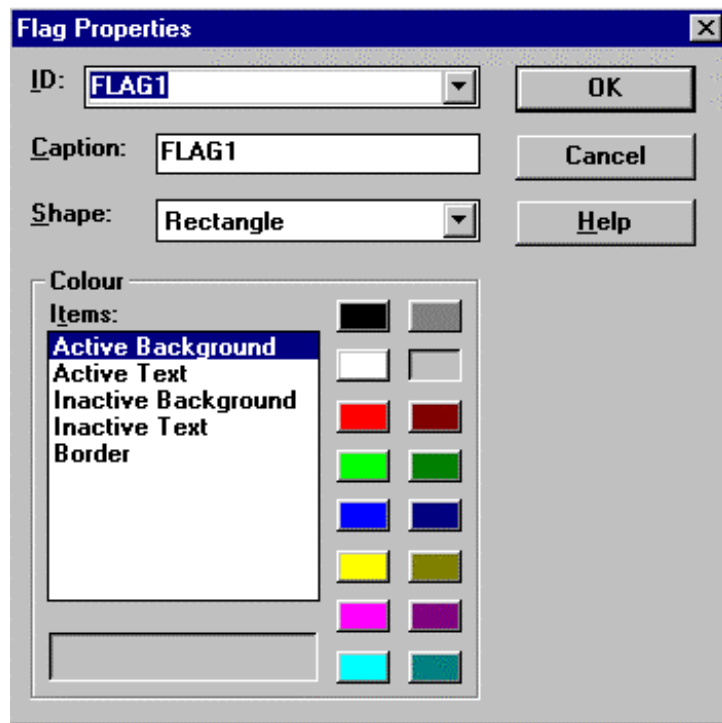
Edit box

This option allows text to be entered into the test panel and text output to be shown in the test panel. This mechanism allows the user to input text and numbers which may be required to trigger events in the script file. Output generated by events occurring in the firmware may also be shown, for example, displaying the contents of a location when a breakpoint is reached in the firmware. The script file can be set up to monitor the breakpoint and trigger an event which will display the required locations when this occurs.

Properties

Each item in a test panel has a number of properties associated with it, which may be specified or changed using the 'Panel Properties' menu option or by double-clicking on the item after the 'Move' option in the test panel edit palette has been selected.

An example of the Flag Properties dialog box is given below. Please note that the Shape and Colour property options are not included for Text and Button items.



The properties which can be specified are given below:

Property **Meaning**

Identity	The 'Identity' property allows the item to be linked to statements given in the script file. For example, the script file will use the 'Identity' to test panel flag values when actioning events.
Caption	The item 'Caption' allows the user to give the item a caption which is displayed in the test panel.
Shape	The shape of flag item.
Colour	The colours of the components of the flag items
File Name	The name of a bitmap image file
Stretch to Fit	Option to specify that a specified bitmap image should be stretched to fit into a defined area on a test panel

Source Debugging

Overview

There are several facilities provided by Micropak 870C to aid source debugging. These facilities are as follows:

- Quick Watch. Displays snapshot information about a selected object or structure.
- Watch. Displays constantly updated information about objects.
- Locals. Displays constantly updated information about local variables currently in scope.
- Call Stack. Displays the current call stack.
- Registers. Displays constantly updated register contents.
- Browser. Shows all function names and source file names.

With the exception of 'QuickWatch' any combination of these functions may be run simultaneously in a test environment. Each of these facilities is described in this section.

Source Windows

MP870C allows you to open one or more views of the source code you are debugging. This allows you to watch the progress of execution through different parts of your code and also allows you to edit the code held in different modules directly.

Whenever code is executing in 'Slow Mode' the yellow PC bar will track the execution of code in a source window. The PC bar will move between any open source windows. If a new window is required to show the PC bar, MP870C will automatically open it and display it on top of the previously activated source window.

The current window focus may be set by the user either into a source window or into any of the non-source windows. If the focus is set in a source window, MP870C will automatically change the current window focus between open source windows so that the focus or active source window is changed when the PC changes. If the focus is not set in a source window, the focus will not be changed. MP870C will however place the source window currently showing the PC on the top of any 'Z' ordered source windows open, so that the user may view the currently executing source.

To tidy the display the user may choose 'Stack Source Windows' from the Windows menu. This will place all open source windows in a Z order with the active window on top.

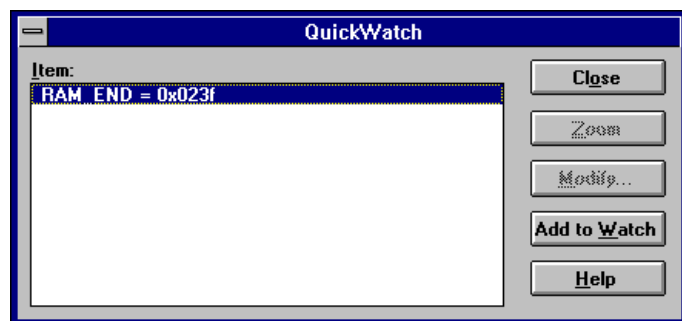
You may edit any of the source windows, however please note that changes to the source code will not become effective until a project rebuild is made.

Quick Watch

The purpose of the 'Quick Watch' facility is to enable you to quickly inspect a variable or type. This is achieved by moving the cursor into the variable name to be interrogated then selecting 'QuickWatch' from the 'Debug' menu or clicking the 'QuickWatch' tool bar icon:



When selected, a window displaying information about the selected item is displayed:



To expand an object which refers to a structure such as a pointer, click the 'Zoom' button.

If the variable is of simple type it may be modified by clicking the 'Modify' button. The 'Modify Variable' dialog box will be displayed which allows a new value for the variable to be input.

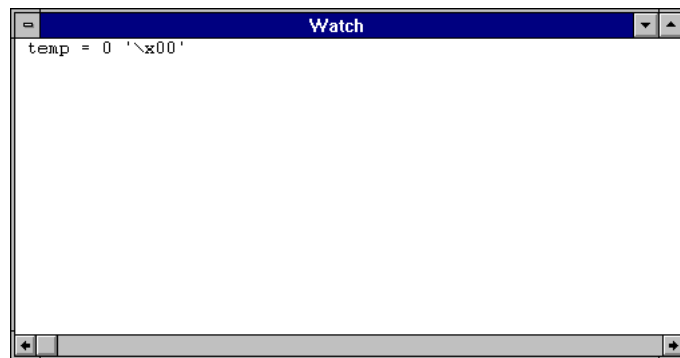
If you wish to subsequently monitor an item on an ongoing basis, the 'Add to Watch' button may be clicked to transfer the item to the 'Watch' window and simultaneously close the 'QuickWatch' window.

Note that the variable being inspected must be in scope.

Watch

This facility allows the ongoing viewing of items while they are being constantly updated by the executing program. It is selected by either choosing 'Watch' from the 'Window' menu or by clicking the 'Add to Watch' button from the 'QuickWatch' menu.

In either case a window similar to the following is displayed:



Any items transferred from the 'QuickWatch' facility will be displayed at the top of this window.

Adding items to the 'Watch' window

Further items can be added to the 'Watch' window in one of two ways:

- By transferring other items from 'QuickWatch' dialog box.

- By typing the name of the item to be monitored.

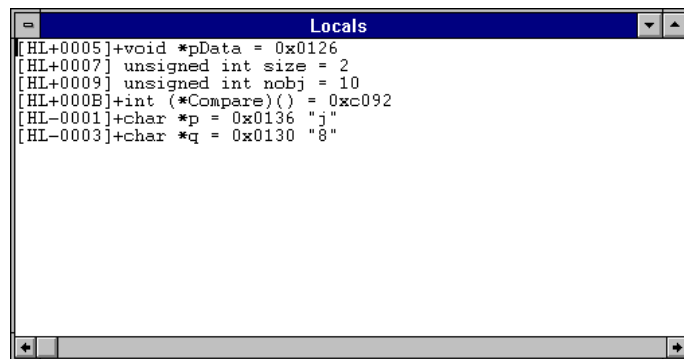
This latter method is accomplished by clicking the edit cursor into the next empty line on the 'Watch' window (i.e. either the first line of the display area or the blank line below the last item already displayed) and pressing the 'Enter' key. The text is checked and if it is a valid item, the value is displayed and updated accordingly. If the item is invalid the text '<undefined>' will be displayed adjacent to it.

Locals

The 'Local' function displays constantly updated information about local variables currently in scope. When selected this facility will continuously update the 'Locals' window, both with the variables in scope and their respective values.

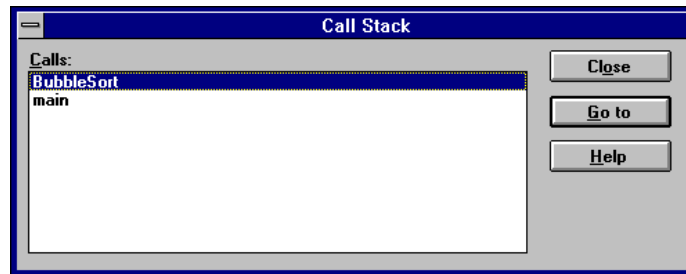
Note that, in addition to displaying the variable name and its current value, the variable type is also shown.

The 'Locals' window is displayed by selecting 'Locals' from the 'Window' menu:



Call Stack

This window, when selected, displays the function call stack of the current program. To view the call stack select 'Show Call Stack' from the 'Debug' menu. A window similar to the following is displayed:

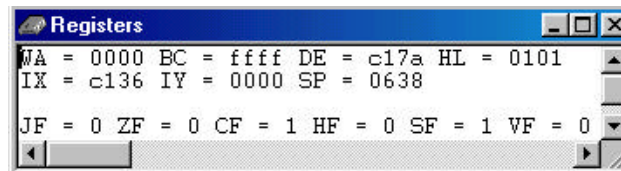


The most recently called function is displayed at the top of the window and the oldest function is displayed at the bottom. The currently active function is highlighted which allows you to assess the local variables which can be accessed from the stack frame for debugging.

From this window you can go to the source file required by clicking the 'Go to' button. If the highlighted function is the most recently called the cursor will be placed in the source file at the current PC position. For other functions the cursor will be placed in the source file at the point at which control returns to the function.

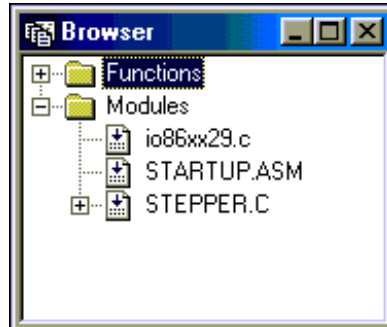
Registers

The contents of the registers can be monitored by selecting 'Registers' from the 'Window' menu. A window similar to the following is displayed:



Browser

Browser information showing function names and all modules are displayed in this window. Clicking on any item will automatically open the source file and place the cursor over any specific function selected.



Using Script Files

Overview

Micropak 870C can process sets of commands contained in script files. These commands can set or check conditions in the simulated system and can be used to run tests automatically or to mimic the behaviour of external hardware.

The command format resembles a simple high level language such as BASIC. The functions which are provided are chosen to simplify the selection of predefined tests or the simulation of external events or items.

Test panels may be used both to display the results of the script processing events and pass input, via user action, to the script processing function.

Script files - Purpose and uses

Script files allow users to extend and customise the simulation facilities and in particular to:

- Extend the simulation by providing automatic handling of external events. The script file facilities allow the behaviour of external hardware and other devices to be simulated automatically.
- Set up semi-automatic control of testing sessions. This allows the running of repetitive or regression tests semi-automatically, by setting up RAM data, running code sessions and checking results.

Test panels and their relation to script files

Test panels are used to provide a user defined, and application specific, user interface to the program under test via script files facilities. They allow, for example, port lines to be controlled by simple 'point and shoot' actions whilst the target code is running.

Examples of script file and test panel uses

The flexibility and power of the script file and test panel combinations allow them to be used in many differing ways, for example:

- Simulating the relationship between triac devices and a delayed thermistor reading in a heater control application, using test panels to display the drive, critical temperatures and effective power applied. Script files could be used to model the changes in performance and relations between the peripheral elements associated with a range of mains voltages or thermal characteristics. They could also show the average main voltage at switching points - allowing the accuracy of zero-cross switching algorithms to be assessed.
- Simulating a DC motor. The script file could be used to relate acceleration to drive signals and relate speed to back-emf, etc., building up to an accurate model of a physical motor.
- Simulating custom LCDs with simple test pattern flags.
- Modelling membrane keypads or other switch inputs, including switch bounce and other non-ideal characteristics.
- Generating serial bit streams such as asynchronous data at differing baud rates to test firmware implemented UART decoding.
- Simulating the performance of firmware implemented ADCs by modelling the functions of external comparators/ RC networks, etc.
- Checking zero cross timing handling by generating a range of waveforms or mixtures of waveforms on the associated pins, etc.
- Modelling the known characteristics of real sensors (such as pulse generating flow sensors). Test panels would allow the user to set and display flow rates, whilst the script files could convert this to a pulse train at an appropriate frequency.
- Simulating inertia of stepper motors, therefore detecting conditions where the firmware ramps the drive speed too quickly.

It can be seen that the applicability of the test panel and script file combination is very wide ranging. It is particularly advantageous in that extensive testing can be performed in the simulated environment before any hardware is ready, or when the consequence of generating incorrect control signals in the real environment would be undesirable. The environment is also very suitable for theoretical and repeatable testing, allowing, for example, one control algorithm to be tested against another in standard conditions.

Module testing is an activity for which the simulated environment is ideal, allowing data tests to be set up easily, and run semi-automatically under the control of an appropriate test panel.

The script language

The grammar and syntax

Here is a summary of the main features of the script file grammar and syntax. For a full, formal definition in modified Backus-Naur Form, see the script file syntax shown as an Appendix.

Here is an informed, narrative description of the grammar and syntax:

Statements and lines

Script files are made up of statements.

- Statements must end with a terminator which can be either a normal end of line (i.e. line feed or carriage return/line feed pair), or a colon character, ':'
- Null statements (such as blank lines) are allowed
- Most statements follow a simple logical format, which is very similar to BASIC

Here is an example of a single line statement:

let SignalVolts = sin(TimeBase)

Here is another example showing two statements on a single line, separated by the ':' character:

let A = B + C : let D = D * E

Elements of script file statements

Script statements are composed from a number of different elements. These are as follows:

Keywords	Identify statement types (such as 'if', 'poke', 'goto', etc.), built-in functions (such as 'sin', 'cos', 'atan', etc.) and qualifiers (such as 'button', 'pin', 'edit'). Note that keywords must be in lower case.
Variables	Identify user data variables. Examples include 'MyVar', 'SignalVolts', 'timebase' and 'StartButtonId\$'. Variables can be integers, real or strings.
Operators	These allow variables and functions to be combined where necessary to form expressions, such as 'Scale * cos(timebase)', where the '*' is the multiplication operator.

These elements are described in more detail in the sections which follow.

Note on comment delimiters

The 'rem' statement allows the insertion of comments. Because it is a statement it must be preceded by a terminator to separate it from any other preceding statements on the same line.

The apostrophe character, ', can be used anywhere on a line to introduce a comment and does not need a preceding terminator.

In both cases the comment is considered to last until the next end of line and comments can therefore include ':' characters.

Script file variables

Internal script file variables are required in order to control and action events. The table below lists the type of variables available:

Type	Description
Integer	Holds integer values. The variable name is suffixed with the '%' character (e.g. 'count%').

Real	Holds floating point values (e.g. 'value').
String	Holds string values. The variable name is suffixed with the '\$' character (e.g. 'name\$').

Variable identifiers

The following set of rules is applied to variables:

- Variables are named by the user
- They need not be declared before use
- There is no significant limit on the length of variable identifiers
- They cannot be reserved words.

The Micropak 870C script file processor is case-insensitive, so, for example, 'SIGNAL' and 'signal' will be treated as the same variable.

Examples of variables

Name	Notes
SignalVoltage	A real (i.e. floating point) because it has no '%' or '\$' suffix.
ButtonName\$	A string (has a '\$' suffix).
SignalVolts%	An integer (i.e. a 32-bit signed number) because it has a '%' suffix.

Numeric type conversions

Like BASIC, the script facilities provide automatic type conversion, as and when appropriate, between Real and Integer values. These types can be mixed in numeric expressions and built-in functions, such as 'abs(<numeric expression>)' or 'cos(<numeric expression>)', can be called with either type or a mixed numeric expression as a parameter.

Script operators and expressions

Here is an informal list of the various operators supported by the script file processor when evaluating expressions. A formal definition of the full syntax of the script language is in the Appendix.

Item	Purpose
*	Multiplication operator
/	Division operator
>	Relational greater than operator
>=	Relational greater than or equal to operator
<	Relational less than operator
<=	Relational less than or equal to operator
=	Relational equal operator
<>	Relational not equal operator
+	Addition operator
-	Arithmetic negation operator and subtraction operator
^	Exponentiation operator (raise to the power)
and	Logical and operator
eqv	Logical equivalence operator
imp	Logical implication operator
not	Logical complement operator
or	Logical inclusive-or operator
xor	Logical exclusive-or operator

Script operator precedence and associativity

Operator	Description	Precedence	Associativity
^	raise to power	0	left
-	unary minus	1	right
*	multiplication	2	left
/	division	2	left
+	addition	3	left
-	subtraction	3	left

>	greater than	4	left
>=	greater than or equal	4	left
<	less than	4	left
<=	less than or equal	4	left
=	equal	4	left
<>	not equal	4	left
not	logical not	5	right
and	logical and	6	left
or	logical or	7	left
xor	logical exclusive-or	8	left
imp	logical implication	9	left
eqv	logical equivalence	10	left

Example expressions

let A = B + C * D + Param ^ Power : rem real numbers

let A\$ = NICK\$ + NAME\$: rem string concatenation

Note - String Expressions

For string expressions, '+' (concatenation) is the only supported operator.

Script file execution and control flow

Files built from statements

Script files are divided into executable statements.

Events start script execution from labelled entry points

Within a script file labels show entry points at which the script file actions can commence. In operation, the execution of the commands in the script file is triggered by 'events'. These events can be user key strokes, script file breakpoints, or the passage of simulated real-time in the target system.

When a nominated event occurs, execution of the script file commands is started at the labelled entry point associated with that event.

Control flow

Execution normally follows sequentially from the first statement to the next statement in the file, however, control transfers using 'goto label' and 'gosub label' statements are possible.

Execution always begins at a labelled entry point and continues until an 'end' statement is reached. If the Micropak 870C simulation was executing code when the originating event occurs, it is suspended whilst any script file segment is still running.

When all segments have run to completion, the simulator execution is resumed.

More on script file events

Script files can be driven by events. An event trigger must first be set up to drive an event. There are five different types of trigger, these being 'breakpoint', 'button', 'edit', 'pin' and 'timeout'. Each event, and how to set up its trigger, is described in the subsequent pages.

All event handlers have a similar format. A simple code skeleton would look something like this:

```
event:  
commands...  
end
```

The label 'event' is the label associated with the event trigger. The 'end' command goes at the end of any other commands in the event handler. This indicates that the event has finished.

Breakpoint events

These events occur when breakpoint conditions, similar to the ones that can be defined in the ‘Breakpoints’ dialog box, are met. These breakpoints are not, however, displayed in the dialog box nor do they stop execution of the simulator. Instead, they cause a section of the script file to be run. The simulator can be stopped from here, if necessary, by executing a ‘stop’ command.

Breakpoint event example

The following piece of code sets up a location breakpoint event trigger at the address of the function call to SetMotorPort. This is on line 197 in the file “main.c”. The location breakpoint is hit whenever the PC reaches this location (indicated by the value ‘1’ for the count argument). The script file code at label “breaklabel” is run when this occurs. The integer variable ‘bp1%’ holds the event number for the breakpoint. This is used when the breakpoint is no longer needed and can be deleted.

```
bp1% = breakpoint(0, "main.c!197", "", 0, 1)
```

The following code will action the breakpoint event trigger ‘bp1%’:

```
on event(bp1%) run breaklabel
```

```
on event(bp1 %) on
```

For more information refer to the descriptions for the ‘on event’ and ‘event’ commands and the ‘breakpoint()’ function.

Button events

These occur when buttons are pressed and released in test panels.

Button event example

The following piece of code sets up a button event trigger within a test panel with ID ‘start’. Note that a test panel containing such a button need not exist when the script file is run. This is because the links between the script file and the button are only made when the instruction itself is run.

```
start% = button("start")
```

Identifies the button ‘start’ with the variable start%.

```
on event(start%) run startlabel
```

event(start%) on

Edit events

These occur when edit boxes in test panels are modified and then lose their input focus. Note that if a script file updates an edit box through an edit command then any modifications will be lost.

Edit event example

The following piece of code sets up an edit box event trigger on an edit box within a test panel with ID ‘duration’. Note that a test panel containing such a button need not exist when the script file is run. This is because the links between the script file and the button are made only when the instruction itself is run.

a% = edit("name")

Returns a string holding the contents of the test panel edit box with identity “name”.

on event(a%) run durationlabel

event(a%) on

Pin events

These occur when there is a voltage change at a pin.

Pin event example

The following piece of code sets up a pin event trigger on a pin in the processor. ‘SCK%’ is an integer variable defining the pin number of the serial clock (SCK) pin.

a% = pin(SCK%)

on event(a%) run event

event(a%) on

Timeout events and timeout event example

The following piece of code sets up a timeout event trigger. The event will occur after “TIME” seconds has elapsed in simulated time. In the example given, “TIME” is a real variable:

```
time% = timeout(TIME)  
on event(time%) run timelabel  
event(time%) on
```

Identities in script files

There are numerous script commands and functions that make references to objects in test panels: the commands are ‘setedit’, ‘setflag’; the functions are ‘button()’, ‘edit()’, ‘getedit()’ and ‘getedit\$()’.

Objects in test panels are referenced through identities. These are strings which identify and name test panel objects. In the commands and functions these are given as simple string expressions.

Links to test panel objects are made at run time. Therefore, when a script file is executed it is only necessary for the test panel object to exist if a command to which it is referenced is encountered. If by mistake an object does not exist then an error message is displayed.

An object in a test panel which can be referenced from a script file has an identity (ID) associated with it. When an object is created in a test panel it is given a default identity. This can be re-defined by displaying the properties dialog box for that object, and modifying the text in the ID edit box. It is recommended that IDs are customised for each object.

Example

```
start% = button("start")
```

Identifies the button “start” with the variable start%.

```
on event(start%) run startlabel  
event(start%) on
```

These commands add a button event trigger and turn it on. When the test panel button with identity ‘start’ is pressed (and then released) it will cause the script code at label ‘startlabel’ to be executed.

```
event(start%) kill
```

Deletes the button event trigger from the button in the test panel with identity 'start'. This in effect kills off the event trigger which was added in the first example.

setedit "duration", 4

Sets the contents of the test panel edit box with identity 'duration' to '4'. Note that the second argument to the edit command can be an expression of type integer, real or string.

setflag "pulse", 1

Sets the test panel flag with identity "pulse" to the highlighted state. The second argument to the flag command is a numeric expression. The flag is highlighted when the expression evaluates to any value other than zero.

a% = getedit("duration")

Returns the numeric value of the test panel edit box , 'duration'. It returns a real argument. In the example above it is type converted to an integer.

a\$ = getedit\$("name")

Returns the string value of the test panel edit box with identity 'name'.

a% = edit("duration")

on event(a%) run durationlabel

event(a%) on

Adds an edit box trigger. When the test panel edit box with identity 'duration' is modified and the input focus lost, the code at label 'durationlabel' will be executed. Note that if a script file updates the edit box through the edit command then any modifications will be lost.

event(a%) off

Turns off the edit box event trigger but does not remove it from the edit box.

Script keywords

These keywords define script statements, qualifiers and built-in script functions. They are described more fully in the following sections.

For a formal definition of the full syntax of the script language, see the script file formal syntax description given as an Appendix.

Item	Purpose
abs	Script function - returns the absolute value of a numeric expression.
acos	Script function - returns the arc cosine of a numeric expression.
asc	Script function - returns ASCII value of a character.
asin	Script function - returns the arc sine of a numeric expression.
atan	Script function - returns the arc tangent of a numeric expression.
breakpoint	Script function - creates a breakpoint event.Returns an event number.
button	Script function - creates a test panel button event.Returns an event number.
chr\$	Script function - returns ascii character of a given numeric value
close	Close a file stream.
connect	Script statement - connects/disconnects Thevenin network to pin.
cos	Built-in script function - returns the cosine of a numeric expression.
dim	Dimension statement to set dimensions for a script array.
edit	Script function - creates a test panel edit box event. Returns an event number.
else	Used with 'if'.
elseif	Used with 'if'.
end	Script statement - specifies the end of an event.
endif	Used with 'if'.
event	Script statement - provides control for breakpoint trigger events.
event on	Turns on a breakpoint trigger events.
event off	Turns off a breakpoint trigger event.
event kill	Delete a breakpoint trigger event.
exit	Exit from the script file.
exp	Built-in script function - returns the value of an exponential function with 'e' as its base.
for	Script statement - specifies the start of a for loop in the script file.
getedit	Script function - allows data to be returned from, a test panel edit box. Returns a value from a test panel edit box.
getedit\$	Built-in script function - returns strings from a test panel edit box.
getfc	Built-in script function - returns the main clock value.
Item	Purpose
getfs	Built-in script function - returns the system clock value.
getrr	Built-in script function - returns register value eg PC,HL,IY,SP,WA,IX.
gettime	Built-in script function - returns the simulation time.
getv	Built-in script function - returns (gets) the voltage value at a pin.
global	Script function - returns the value of a global C variable.

go	Script statement - starts the execution of the target code.
gosub	Script statement - performs an event in the script file as a subroutine and returns here on encountering a return or an end statement.
Goto	Script statement - transfers script control to the label specified as the destination.
If	Script statement - conditional statement allows the behaviour of the script file to be adapted automatically to actions and states that occur in the target system.
input	Script statement for inputting data from a file.
Left\$	Script function - returns the left n characters of a string.
Len	Script function - returns the length of a string.
Let	Script statement - assignment.
Local	Script function - returns the value of a local C variable.
Mid\$	Script function - returns the middle part of a string.
Next	Used with 'for'.
On event	Script statement - conditional statement to allow parts of script files to be run conditionally.
run	Script statement for opening a file.
Open	Built-in script function - returns a value from target memory.
Peek	Script statement - puts a value to target memory.
Poke	Statement qualifier, used with 'add' and 'delete'.
Pin	Script statement for writing data to a file.
Print	Script statement - introduces a comment.
Rem	Script statement - used with 'until' for control loops.
Repeat	Script statement - specifies the end of a script file subroutine.
Return	Script function - returns the right part of a string.
Right\$	Script statement - specifies a bitmap file to be displayed
Setbitmap	Script statement - allows data to be output to a test panel edit box.
setedit	Script statement - allows data to be output to a test panel flag box.
Setflag	Script statement - sets the target register to a given value eg PC,HL,IY,IX,SP,WA
Setrr	Script statement - sets the resistance value for the external Thevenin network of a pin.
Setr	Script statement - sets the voltage generator value for the external Thevenin network of a pin.
Setv	
Item	Purpose
sin	Built-in script function - returns the sine of a numeric expression.
sgn	Built-in script function - returns the sign of a numeric expression.
sqr	Built-in script function - returns the square root of a numeric expression.
step	Used with 'for' statement to define the loop increment.
stop	Script statement - stops the execution of the target code.
tan	Built-in script function - returns the tangent of a numeric expression.

then Used in 'if' statements.
timeout Script function - creates a timeout event. Returns an event number.
until Used with 'repeat'.

Note that some script operators are also reserved words (e.g. and, xor, etc.).

Script file commands and functions - detailed descriptions

This section describes the various script file commands and functions in detail. They are given in alphabetical order.

ABS

Script file function. Returns the absolute value of a numeric expression.

Format

abs(<value>)

value input parameter (numeric expression)

Example

a = abs(-1.6) 'would return 1.6

ACOS

Script file function. Returns the arc cosine (in radians) of a numeric expression.

Format

acos(<value>)

value input parameter (numeric expression)

Example

a = acos(1) 'would return 0

AND

Logical and operator. The and operator performs a bitwise operation on two integer operands.

The truth table for the operator is:

X	Y	X and Y
1	1	1
1	0	0
0	1	0
0	0	0

Example

a% = b% and c%

ASC

Script file function. Returns the ASCII value associated with a character.

Format

asc(<char>)

char input parameter (character)

Example

a% = asc("A") 'would return 65

ASIN

Script file function. Returns the arc sine (in radians) of a numeric expression.

Format

asin(<value>)

value input parameter (numeric expression)

Example

a = asin(1) 'would return pi / 2

ATAN

Script file function. Returns the arc tangent (in radians) of a numeric expression.

Format

atan(<value>)

value input parameter (numeric expression)

Example

a = atan(1) 'would return pi / 4

BREAKPOINT

This is a function which allows the details of breakpoints to be set in script files.

Breakpoints set in this way act as event triggers. The specification of breakpoints in this function allow a similar range of options to those provided by the user breakpoint facility in the simulator (which is accessed via the Breakpoints dialog box).

When the breakpoint is reached the event handler associated with the breakpoint will be triggered.

The value returned by the function is an event number which identifies the breakpoint event. This should be stored for use later on when the event is no longer needed, and can be deleted.

Format

breakpoint(<type>, <location>, <expression>, <length>, <count>)

type Breakpoint-type
 0 - location reached

	1 - location reached and given expression true	
	2 - location reached and given expression changed	
	3 - expression true	4
	- expression changed	
location	A program location or address.	
	A numeric ROM address may be given.	A
	source line number of the active source file may be given	
	prefixed with a '.'	A
	file name followed by a '!' and then a line number may be	
	given.	
expression	Any L-Value expression may be given.	
length	The length of the data in the expression may be given.	
Counter	An integer expression or value specifying the number of times the break condition should be reached before an event is triggered.	

Examples

This first example creates a location breakpoint event. The condition to trigger the event will be a location condition occurring (1) times, when the PC reaches line 100 in the source file "coreb.c". Note that the expression and length fields contain dummy values ("" and 0). For this type of breakpoint they are not used. The breakpoint event number is stored in bp1% for use later on when the event needs to have a label associated with it, be turned on, off or deleted (see the EVENT statement).

```
bp1% = breakpoint(1, "coreb.c!100", "", 0, 1)
```

This second example creates an l-value changed breakpoint event. The condition to trigger the event is a modification to the l-value "myvar". Since myvar is not an array we specify 1 in the length field. Had it been, then we could have specified 1, 2... up to the number of elements in the array. The count field is set to 1, which means that we want the event to occur every time the l-value is modified.

```
bp2% = breakpoint(1, "", "myvar", 1, 1)
```

Again, the breakpoint event number is stored, this time in bp2%, for use later on.

CHR\$

Script file function. Returns the ASCII character associated with a numeric value

Format

chr\$(<value>)

value input parameter (numeric expression)

Example

a\$ = chr\$(65) 'would return "A"

CLOSE

Close a file.

Format

close [#]<file-number>

file-number a valid file number (numeric expression) as obtained from the open statement.

Example

close #1

CONNECT

Connects/disconnects Thevenin network to a pin.

Format

connect <pin-number>, <connect, disconnect>

pin-number a valid pin number (numeric expression)

connect, disconnect value 1 or 0 to show connected or disconnected (numeric expression)

Example

```
connect startpin%, 1
```

COS

Script file function. Returns the cosine of a numeric expression (in radians).

Format

```
cos(<value>)
```

value input parameter (numeric expression)

Example

```
a = cos(2 * 3.1415926 / 6) 'would return (approximately) 0.5
```

DIM

Script file declaration. Declares the dimensions of a script variable array.

Format

```
dim <variable>(<dimension>{, <dimension>...})
```

variable the array variable

dimension a dimension of the array.

Example

```
dim display$(10) 'would dimension an array of 10 strings
```

The statement to access elements in the array is:

```
name$ = display$(3)
```

EDIT

Specifies that a breakpoint trigger event is to be set on a test panel edit box. This is a function which returns an event trigger number.

Format

edit(<identity>)

identity name of a test panel edit box

Example

```
flagvar% = edit("DISP")
```

```
on event(flagvar%) run toggle
```

```
event(flagvar%) on
```

END

Specifies the end of an event. Control will flow through events in the script file until this command is reached.

Example

```
end
```

EQV

Logical equivalence operator. This operator performs a bitwise operation on two integer operands. The truth table for the operator is:

X	Y	X eqv Y
1	1	1
1	0	0
0	1	0
0	0	1

Example

```
a% = b% eqv c%
```

EVENT

Allows event triggers to be turned on or off or deleted.

Event trigger actions are specified by setting the value from an event function to a script file variable. The script file variable is then used to specify the event. The possible event functions are breakpoint, button, edit, pin and timeout. You should refer to the relevant section of this manual for details of the functions.

The action routine to be run when the event occurs is specified by the ON EVENT statement.

Both the button and edit formats below refer to objects in test panels. A button event is triggered when a test panel button is pressed and then released. An edit event is triggered when an edit box in a test panel is modified, and the edit box then loses the input focus. Note that if a script file updates the edit box then any modifications will be lost. The pin format refers to the pin on the processor. A pin event is triggered whenever there is a voltage change at the pin in question.

Format

event(<identity>) <status>

identity the identity of the event a script file variable

status on - turn the event on
 off - turn the event off
 kill - remove the event

Examples

timer% = timeout(0.5)

on event(timer%) run toggle

event(timer%) on

press% = button ("start")

on event(press%) run toggle

event(press%) on

EXP

Script file function. Returns the value of an exponential function with 'e' as its base.

Format

exp(<value>)

value input parameter (numeric expression)

Example

a = exp(2) 'would return 'e' raised to the power 2

FOR - TO - [STEP]

Specifies the start of a for loop in the script file. This allows commands in a script file to be repeated a specified number of times.

Format

for <control> = <start> to <end> [step <increment>]

control an internal variable which will be incremented each time around the loop

start the starting value to be set (numeric expression)

end an ending value against which the contents of the control variable will be checked. When the control variable value matches the end value, the for loop will be terminated (numeric expression)

increment an optional step increment to be added to the control variable (numeric expression)

Example

for count% = 0 to 10

for count% = start% to end% step 2

GETEDIT(\$)

Script file function. Allows data and strings to be returned from a test panel edit box.

Format

getedit(<identity>)

getedit\$(<identity>)

identity input parameter (string expression)

Example

```
a$ = getedit$("string")
```

GETFC

Read the frequency of the main system clock.

This statement may not be used on its own but must be used as an expression within other statements.

Format

getfc()

Example

```
clock = getfc()
```

GETFS

Read the frequency of the sub-system clock. This statement may not be used on its own but must be used as an expression within other statements.

Format

getfs()

Example

```
subclock = getfs()
```

GETrr

Read the current value of the specified register. This statement may not be used on its own but must be used as an expression within other statements. Valid registers are PC,SP,HL,IX,IY,WA

Format

getpc()

Example

```
reg% = gethl()
```

GETTIME

Obtain the currently held simulation time. This time is expressed in seconds. This statement may not be used on its own but must be used as an expression within other statements.

Format

gettime()

Example

```
simtime = gettime()
```

GETV

Read the voltage on a pin. The voltage read is expressed in volts.

This statement may not be used on its own but must be used as an expression within other statements.

Format

getv(<pin-number>)

pin-number a valid pin number (numeric expression)

Example

```
setv startpin%, getv(Gnd%)
```

VOLTS = getv(9)

GLOBAL

Returns the value of a global variable in the program under simulation. The l-value given must resolve to a simple type, either an integer or floating point value. If it does not, or the l-value is invalid then zero is returned. You can use the C dereference operator *, or the member access operators . and ->, when specifying the l-value.

This statement may not be used on its own but must be used as an expression within other statements.

Format

`global(<l-value>)`

l-value the global variable to retrieve (numeric expression)

Example

a% = global("myvar")

GO

Starts the execution of the target code. There are no parameters.

Example

go

GOSUB

Perform an event in the script file as a subroutine and return here on encountering a return statement or an end statement.

Format

`gosub <label>`

label the label of the event to be performed

Example

```
gosub show
```

GOTO

Transfers control to the label specified.

Format

```
goto <label>
```

label the label of the destination

Example

```
goto show
```

IF - THEN - [ELSEIF] - [ELSE] - ENDIF

A conditional statement which assesses the value of an expression and performs a set of statements if the condition is found to be true.

Optionally, a different block of statements may be performed if the expression is not found to be true. Statement blocks are terminated by the ELSEIF, ELSE or the ENDIF, whichever is appropriate.

Format

```
if <expression> then <statement block> {elseif <statement block>} [else  
  <statement block>] endif
```

expression which can be evaluated to true or false (numeric expression)

statement block one or more statements in a block. The block is terminated by either
elseif, else or endif

Example

```
if getv(driv1%) < threshold then  
  setflag "MOTOR1", 0  
elseif
```

```
setflag "MOTOR1", 1
endif
```

IMP

Logical implication operator which performs a bitwise operation on two integer operands. The truth table is as follows:

X	Y	X imp Y
1	1	1
1	0	0
0	1	1
0	0	1

Example

```
a% = b% imp c%
```

INPUT

Obtains a value from a file.

Format

```
input #<file>, <item>[, <item>...]
```

file number of the file as given on open

item name of a script file variable

Example

```
input #1, driv1%
```

LEFT\$

Script file function. Returns string value which is the left part of the given string.

Format

left\$(<string>, <length>)

string input parameter (ASCII string)

length length of the returned string

Example

name\$ = left\$("Henry Bloggs", 5) 'would return "Henry"

LEN

Script file function. Returns a numeric value which is the length of a given string.

Format

len(<string>)

string input parameter (ascii string)

Example

length% = len("Henry Bloggs") 'would return 12

LET

Assigns a value to an internal script file variable. The LET is optional.

Format

[let] <item> = <value>

item name of a script file variable (string expression)

value input parameter (numeric expression)

Example

let driv1% = 9

LOCAL

Returns the value of a local variable in the program under simulation. The l-value given must resolve to a simple type, either an integer or floating point value. If it does not, or the l-value is invalid then zero is returned. You can use the C dereference operator *, or the member access operators . and ->, when specifying the l-value.

This statement may not be used on its own but must be used as an expression within other statements.

Format

local(<l-value>)

l-value the local variable to retrieve (numeric expression)

Example

```
a% = local("myvar")
```

MID\$

Script file function. Returns string value which is the middle part of the given string.

Format

mid\$(<string>, <start>, <length>)

string input parameter (ASCII string)

start input parameter start position in the string

length length of the returned string

Example

```
name$ = mid$("Henry James Bloggs", 7, 5) 'would return  
"James"
```

NOT

Logical not operator which performs a bitwise operation on an integer operand.

The truth table for this operator is as follows:

X	not X
1	0
0	1

Example

a% = not a%

OPEN

Open a file for capturing data or inputting data.

Format

open <file-name> [for <mode>] [access <access>] [<lock>] as
[#]<expression>

file-name name of the file to be opened

mode for mode type which is one of :-

append	input
output	

access	read [write]	write
---------------	--------------	-------

lock	shared	lock
	read [write]	lock
	write	

expression numerical expression between 1 and 255 to identify the file

Example

open "serdata.txt" for input access read as #1

open "test1.txt" for append access write as #2

OR

Logical or operator which performs a bitwise operation on two integer operands. The truth table is as follows:

X	Y	X or Y
1	1	1
1	0	1
0	1	1
0	0	0

Example

a% = b% or c%

PEEK

Function which gets a value from target memory.

Format

peek(<memory>)

memory address of memory location (numeric expression)

Example

count% = peek(100)

PRINT

Obtains a value from a file.

Format

print #<file>, <item>[{,|;} <item>...]

file number of the file as given on open

item name of a script file variable (string expression)

Example

```
print #1, driv1%
```

POKE

Writes a value to target memory.

Format

```
poke <memory>, <value>
```

memory address of memory location (numeric expression)

value value to be output (numeric expression)

Example

```
poke 100, 4
```

REM

Comment statement in a script file. This statement can be followed by any textual information. It may be used on an individual line or on lines containing a statement:

Example

```
rem Highlight the flags showing the phases of the stepper
```

Note on comment delimiters

The rem statement allows the insertion of comments. Because it is a statement, it must be preceded by a terminator to separate it from any other preceding statements on the same line.

The apostrophe character ' can be used anywhere on a line to introduce a comment and does not need a preceding terminator.

In both cases the comment is considered to last until the next end of line, and comments can therefore include ':' characters.

REPEAT - UNTIL

A repeat control loop in which a number of statements are repeatedly performed until the expression given evaluates to true.

Format

repeat <statement block> until <expression>

statement block one or more statements in a block. The block is terminated by until

expression expression which can be evaluated to true or false (numeric expression)

Example

```
repeat
v = getv(driv1%)
until v = threshold
```

RETURN

Specifies the end of a script file subroutine. This statement takes no parameters.

Format

return

Example

```
show:
setflag "on", 1
return
```

RIGHT\$

Script file function. Returns string value which is the rightmost part of the given string.

Format

right\$(<string>, <length>)

string input parameter (ASCII string)

start input parameter start position in the string

Example

**name\$ = right\$("Henry James Bloggs", 6) 'would return
"Bloggs"**

SETBITMAP

Specifies the name of a bitmap file to be displayed and displays the image in the testpanel.

Format

setbitmap <filename>, <data>

identity name of a test panel bitmap item (string expression)

filename name of a bitmap image file (string expression)

fit flag flag 0/1 to show whether or not the image should be stretched to fit the defined space in the test panel or displayed at actual size.

Example

setbitmap "POWER", "pushbutton.bmp", 1

SETEDIT

Allows data to be output to a test panel edit box.

Format

setedit <identity>, <data>

identity name of a test panel edit box (string expression)

data data to be output. Data must be 0 or 1 (numeric expression)

Example

```
setedit "DISPLAY", 0
```

SETFLAG

Allows data to be output to a test panel flag.

Format

flag <identity>, <data>

identity name of a test panel flag box (string expression)

data data to be output. Data must be 0 or 1 (numeric expression)

Example

```
setflag "STOPACT", 0
```

SETrr

Sets up the value in the specified register

This command may be used to set up any register pair including PC,SP,IX,IY,WA,HL.

Format

setrr <value>

value valid rom address for the processor (numeric expression) or
any other value valid for the specific register.

Example

```
setpc 100
```

```
setix 0f
```

SETR

Sets up the value of the external resistance on a pin.

Format

setr <pin-number>, <value>

pin-number a valid pin number (numeric expression)

value resistance value to be set, expressed in ohms (numeric expression)

Example

```
setr startpin%, 10000
```

```
setr 9, 10000
```

SETV

Sets up the value of the external voltage on a pin.

Format

setv <pin-number>, <value>

pin-number a valid pin number (numeric expression)

value voltage value, expressed in volts (numeric expression)

Example

```
setv startpin%, getv(Gnd%)
```

```
setv 9, 5
```

SIN

Script file function. Returns the sine of a numeric expression (in radians).

Format

sin(<value>)

value input parameter (numeric expression)

Example

```
a = sin(3.1415926 / 6) 'would return (approximately) 0.5
```

SGN

Script file function. Returns the sign of a numeric expression.

If the expression is negative then -1 is returned. If the expression is positive then 1 is returned and if it is equal to zero then 0 is returned.

Format

`sgn(<value>)`

value input parameter (numeric expression)

Example

a% = sgn(-10) 'would return -1

SQR

Script file function. Returns the square root of a numeric expression. The expression must be positive.

Format

`sqr(<value>)`

value input parameter (numeric expression)

Example

a = sqr(13) 'would return 3.605...

STOP

Stops the target code execution. There are no parameters.

Example

stop

TAN

Script file function. Returns the tangent of a numeric expression (in radians).

Format

tan(<value>)

value input parameter (numeric expression)

Example

a = tan(3.1415926 / 4) 'would return (approximately) 0.01371

TIMEOUT

Specifies a time-out value which will be monitored to trigger events when the time-out expires. This facility allows the testing of time-dependent routines in the target code.

Format

timeout(<value>)

value value of the time-out in seconds (numeric expression)

Example

time% = timeout(0.010)

XOR

Logical exclusive-or operator which performs a bitwise operation on two integer operands. The truth table is as follows:

X	Y	X xor Y
1	1	0
1	0	1
0	1	1
0	0	0

Example

a% = b% xor c%

Keyboard Summary

Editing keys

A complete list of the keys available for the editing function is given in the chapter 'Using the Editor'.

Accelerator keys

Here is a summary of the 'accelerator' keys used to provide fast access to commonly used menu options:

File menu

Ctrl+N	New file
Ctrl+O	Open file
Ctrl+S	Save file

Edit menu

Ctrl+Z	Undo
Ctrl+A	Redo
Ctrl+X	Cut
Ctrl+C	Copy
Ctrl+V	Paste
Ctrl+F	Find
Ctrl+R	Replace

View menu

F4	Next Error
Shift+F4	Previous Error
Ctrl+F2	Toggle Bookmark
F2	Next Bookmark
Shift+F2	Previous Bookmark

Project menu

Ctrl+F3	Compile File
Shift+F3	Build
Alt+F3	Rebuild All

Test menu

Ctrl+P	Run Script
Ctrl+Q	Stop Script

Debug menu

F5	Go
F6	Step Into
F7	Step Over
F8	Step Out
F9	Step to Cursor
Ctrl+F5	Go and Go
Alt+F5	Stop Debugging
Ctrl+F9	Set PC to Cursor
Ctrl+T	Call Stack
Ctrl+U	QuickWatch
Ctrl+B	Breakpoints

Trace menu

Shift+F5	Go (roll) Back
Shift+F6	Step Back Into
Shift+F7	Step Back Over
Shift+F8	Step Back Out
Shift+F9	Step Back to Cursor
Ctrl+I	Signals Zoom In
Ctrl+J	Signals Zoom Out
Ctrl+K	Snap Signals

Window menu

Ctrl+D	Device Window
Ctrl+E	Pin Window

Breakpoint function

F10	Toggle Breakpoint
------------	-------------------

Appendix A - Script file grammar

Definition

This appendix shows the complete grammar of the script file language. It encompasses a complete definition of the language of the script processor.

It is shown in modified Backus-Naur form. Nonterminal symbols and certain tokens are shown in italics. Keywords, and other tokens are shown in a fixed width, typewriter font. Note that keywords, can be given in both upper and lower case, or a mixture of the two. For the sake of clarity, all keywords are shown here in upper case. The subscript *opt* denotes that the symbol, either nonterminal or token, is optional.

interpreter-unit :

*label*_{opt} *statement*_{opt} *newline*

*label*_{opt} *statement*_{opt} :

Statements

statement :

close-statement

connect-statement

dim-statement

else-statement

elseif-statement

end-statement

endif-statement

event-statement

for-statement

go-statement

gosub-statement

goto-statement

if-statement

input-statement

let-statement

next-statement

on-statement
open-statement
poke-statement
print-statement
repeat-statement
return-statement
setedit-statement
setflag-statement
setrrr-statement
setr-statement
setsp-statement
setv-statement
stop-statement
until-statement

close-statement :
CLOSE *close-list*_{opt}

close-list :
 close-list , #_{opt} *expression*
 #_{opt} *expression*

connect-statement :
CONNECT *expression* , *expression*

dim-statement :
DIM *identifier* (*dim-list*)

dim-list :
 dim-list , *expression*
 expression

else-statement :
ELSE

elseif-statement :
ELSEIF *expression* THEN

end-statement :
END

endif-statement :
ENDIF

event-statement :

EVENT (*expression*) *event-action*

event-action :

ON
OFF
KILL

for-statement :

FOR *identifier* = *expression* TO *expression* *step*_{opt}

step :

STEP *expression*

go-statement :

GO

gosub-statement :

GOSUB *label*

goto-statement :

GOTO *label*

if-statement :

IF *expression* THEN

input-statement :

INPUT # *expression* , *input-list*

input-list :

input-list , *identifier*
identifier

let-statement :

LET_{opt} *identifier* = *expression*

next-statement :

NEXT *next-list*_{opt}

next-list :

next-list , *identifier*
identifier

on-statement :

ON EVENT (*expression*) RUN *label*

open-statement :

OPEN *expression* *mode*_{opt} *access*_{opt} *lock*_{opt} AS #_{opt} *expression*
*length*_{opt}

mode :
 FOR *mode-type*

mode-type :
 APPEND
 BINARY
 INPUT
 OUTPUT
 RANDOM

access :
 ACCESS *access-type*

access-type :
 READ WRITE_{opt}
 WRITE

lock :
 SHARED
 LOCK READ WRITE_{opt}
 LOCK WRITE

length :
 LEN = *expression*

poke-statement :
 POKE *expression* , *expression*

print-statement :
 PRINT # *expression* , *print-list*_{opt}

print-list :
 print-list ; *expression*_{opt}
 print-list , *expression*_{opt}
 *expression*_{opt}

repeat-statement :
 REPEAT

return-statement :
 RETURN

setedit-statement :
 SETEDIT *expression* , *expression*

setflag-statement :

SETFLAG *expression* , *expression*

setrr-statement :

SETIX *expression*

SETIY *expression*

SETHL *expression*

SETPC *expression*

SETSP *expression*

SETWA *expression*

setr-statement :

SETR *expression* , *expression*

setsp-statement :

SETSP *expression*

setv-statement :

SETV *expression* , *expression*

stop-statement :

STOP

until-statement :

UNTIL *expression*

Expressions

expression :

eqv-expression

eqv-expression :

eqv-expression EQV *imp-expression*

imp-expression

imp-expression :

imp-expression IMP *xor-expression*

xor-expression

xor-expression :

xor-expression XOR *or-expression*

or-expression

or-expression :

or-expression OR *and-expression*

and-expression

and-expression :

and-expression AND *not-expression*
not-expression

not-expression :

NOT *not-expression*
relational-expression

relational-expression :

relational-expression > *additive-expression*
relational-expression >= *additive-expression*
relational-expression < *additive-expression*
relational-expression <= *additive-expression*
relational-expression = *additive-expression*
relational-expression <> *additive-expression*
additive-expression

additive-expression :

additive-expression + *multiplicative-expression*
additive-expression - *multiplicative-expression*
multiplicative-expression

multiplicative-expression :

multiplicative-expression * *uminus-expression*
multiplicative-expression / *uminus-expression*
uminus-expression

uminus-expression :

- *uminus-expression*
power-expression

power-expression :

power-expression ^ *unary-expression*
unary-expression

unary-expression :

identifier
identifier (*index-list*)
real-constant
integer-constant
string-constant
ABS (*expression*)

ACOS (*expression*)
ASC (*expression*)
ASIN (*expression*)
ATAN (*expression*)
BREAKPOINT (*expression* , *expression* , *expression* ,
 expression , *expression*)
BUTTON (*expression*)
CHR\$ (*expression*)
COS (*expression*)
EDIT (*expression*)
EXP (*expression*)
GETEDIT (*expression*)
GETEDIT\$ (*expression*)
GETFC ()
GETFS ()
GETIX ()
GETIY ()
GETHL ()
GETPC ()
GETSP ()
GETWA ()
GETTIME ()
GETV (*expression*)
GLOBAL (*expression*)
LEFT\$ (*expression* , *expression*)
LEN (*expression*)
LOCAL (*expression*)
MID\$ (*expression* , *expression* *mid-length*_{opt})
PEEK (*expression*)
PIN (*expression*)
RIGHT\$ (*expression* , *expression*)
SIN (*expression*)
SGN (*expression*)
SQR (*expression*)
TAN (*expression*)
TIMEOUT (*expression*)
(*expression*)

index-list :

expression , index-list
index-list

mid-length :

, expression

Appendix B - Script file example

```

'*****
'* File: stepper.scr
'* Desc:
'* File for demonstration of the MP870/C simulator.
'*****

'*****
'* This integer variable determines whether script file
'* breakpoints should be demonstrated. Note that the
'* breakpoints will stop the execution of the program, and
'* will require you to select the Go command from the Debug
'* menu to start things going again.
'*****

    demonstatebreakpoints% = 0

'*****
'* Allocate vars to pin numbers. By allocating variables
'* with pin numbers, makes the script file easier to
'* understand.
'*****

    driv1% = 61
    'the stepper motor drive lines
    driv2% = 62
    driv3% = 63
    driv4% = 64
    Vdd% = 5
    'the power supply lines
    Gnd% = 1
    startpin% = 27                                'the
lines which the real

```

```

    stoppin% = 26
    'switches are connected to

'*****
'* Set voltages on pins and make connections.
'*****

    setv startpin%, getv(Vdd%)          'both
switches made inactive
    setv stoppin%, getv(Vdd%)          '(high)
    setr startpin%, 10000               'resistance
= 10K
    setr stoppin%, 10000
    connect startpin%, 1                'allow the
script file to
    connect stoppin%, 1
    'change the lines

'*****
'* Assign button IDs to run pieces of script files. So if a
'* button is pressed in a test panel the relevent script
'* commands are processed.
'* STARTBUTTON and STOPBUTTON are the ids given to the panel
'* button. actstart and actstop are the labels for the
'* script file functions.
'*****

    start.event% = button("STARTBUTTON")
    on event(start.event%) run actstart
    event(start.event%) on

    stop.event% = button("STOPBUTTON")
    on event(stop.event%) run actstop
    event(stop.event%) on

'*****
'* Any changes to voltage on the motor drive lines will call
'* the appropriate script routine.
'*****

```

```

driv1.event% = pin(driv1%)           'pin 9
on event(driv1.event%) run show1
event(driv1.event%) on

driv2.event% = pin(driv2%)           'pin 10
on event(driv2.event%) run show2
event(driv2.event%) on

driv3.event% = pin(driv3%)           'pin 11
on event(driv3.event%) run show3
event(driv3.event%) on

driv4.event% = pin(driv4%)           'pin 12
on event(driv4.event%) run show4
event(driv4.event%) on

'*****
'* The following section of script code is to produce a
'* breakpoint which has to be hit 5 times before it stops
'* execution and warns the user in the dialog box of the
'* test panel.
'*****

    if demonstatebreakpoints% then
        bpcount% = 5           'set up initial count

        bcbrk.event% = breakpoint(0,"stepper.c!351", "",
0, 1)

                                'set up the
bp, which when

                                'the
program counter is

                                'about to
hit set_phase_7,

                                'the script
code at pcbrk

                                'is called

        on event(bcbrk.event%) run pcbrk

```

```

        event(bcbbrk.event%) on
    endif

'*****
'* The following breakpoint is one set up for data in which
'* it will check that when Port0 is written to (w) that a
'* phase can be seen on the port, (ie Port 0 <> 0). If Port0
'* could not be written to correctly then it will give the
'* error in the edit dialog box.
'*****

    if demonstatebreakpoints% then
        portbbrk.event% = breakpoint(3, "", "IOP0 >= 0",
0, 1)

        on event(portbbrk.event%) run portbbrk
        event(portbbrk.event%) on
    endif

'*****
'* If the user changes the power supply change the switching
'* threshold.
'*****

    vdd.event% = pin(Vdd%)
    on event(vdd.event%) run adjthes
    event(vdd.event%) on

    gnd.event% = pin(Gnd%)
    on event(gnd.event%) run adjthes
    event(gnd.event%) on

adjthes:
    threshold = (getv(Vdd%) + getv(Gnd%)) / 2
    go                'start the instruction
decoder

    end                'end of processing

'*****

```

```
'* The following script routine is called when the program
'* hits the set_phase_7 breakpoint.
'*****

pcbrk:
    bpcount% = bpcount% - 1                'decrement
bp count
    if bpcount% = 0 then                    'check the count
        stop                               'if 0 stop
execution of

    'instructions, initialise
    bpcount% = 5                           'count back to 5

    'report to the user that

    'the bp has been hit
        setedit "DIALOG", "Breakpoint hit"
    endif
end
'end of this routine

'*****

'* This is a another way to find the bug in the tutorial
'* program.
'*****

portbbrk:
    value% = peek(0)
    if value% = 15 then
        setedit "DIALOG", "Wrote 0FH to port 0"
        stop
    endif
end

'*****

'* Highlight the flags showing the phases of the stepper
'* motor.
'* MOTOR1, MOTOR2, MOTOR3 and MOTOR4 are the ids of the
```

```
'* flags in the panel. A value of 1, highlights the flag, a
'* 0 will turn the highlighting of the flag (segment) off.
'*****

show1:
    if getv(driv1%) < threshold then
        setflag "MOTOR1", 0
    else
        setflag "MOTOR1", 1
    endif
end

show2:
    if getv(driv2%) < threshold then
        setflag "MOTOR2", 0
    else
        setflag "MOTOR2", 1
    endif
end

show3:
    if getv(driv3%) < threshold then
        setflag "MOTOR3", 0
    else
        setflag "MOTOR3", 1
    endif
end

show4:
    if getv(driv4%) < threshold then
        setflag "MOTOR4", 0
    else
        setflag "MOTOR4", 1
    endif
end

'*****
'* Script code called when the Start button is pressed.
'* Checks that the I/O line is inactive (HIGH).
```

```

'* If inactive makes the line active. Shows the new status.
'* After 0.010 seconds (10ms) runs script code to deactivate
'* I/O line
'*****

actstart:
    if getv(startpin%) > threshold then
        setv startpin%, getv(Gnd%)
        setflag "STARTACT", 1

        inactstart.event% = timeout(0.010)    '10ms
        on event(inactstart.event%) run inactstart
        event(inactstart.event%) on
    endif
end

'*****
'* Code to deactivate I/O line and show the new status.
'*****

inactstart:
    setv startpin%, getv(Vdd%)
    setflag "STARTACT", 0

    event(inactstart.event%) kill
end

'*****
'* Script code called when the Stop button is pressed.
'*****

actstop:
    if getv(stoppin%) > threshold then
        setv stoppin%, getv(Gnd%)
        setflag "STOPACT", 1

        inactstop.event% = timeout(0.010)    '10ms
        on event(inactstop.event%) run inactstop
        event(inactstop.event%) on

```

```
endif
end

inactstop:
    setv stoppin%, getv(Vdd%)
    setflag "STOPACT", 0

    event(inactstop.event%) kill
end

'*****
'* End of script file.
'*****
```

INDEX

A

accelerator keys, 53, 67, 157
breakpoints, 159
Debug menu, 158
Edit menu, 157
File menu, 157
Project menu, 157
Test menu, 157
Trace menu, 158
View menu, 157
Window menu, 159

B

batch file, 57, 58
breakpoint, 36, 37
breakpoints, 77, 87
accelerator key, 159
break at location, 87
break at location if expression is true, 87
break at location if l-value has changed, 88
break when expression is true, 88
break when l-value has changed, 88
counter field, 89
disabling, 91
enabling, 91
example, 89
expression/l-value field, 88
length field, 89

location field, 88
removing, 91
script file facilities, 92
setting, 91
type field, 87
viewing current set ups, 89
Breakpoints, ix, 10, 11, 36, 37, 40, 49, 50, 58, 83, 158

C

call stack, 114
check boxes, 52
code coverage, 1, 85
code coverage, clearing, 86
controlling execution, 77
customised memory views, 98

D

Debug
breakpoints, 87
fast mode, 33, 77, 80
options, 77, 80, 84
reset, 79
slow mode, 33, 77, 78, 80
debug information file, 58
Debug menu, 31, 46, 49, 79, 87, 158
go, 32, 37
show call stack, 114
debug mode, 33
device information, 103
dialog boxes, 51
directory structure, 5

E

Edit menu, 46, 47, 157
editing a project, 60
editor
 keyboard functions, 70
 locating and changing text, 71
 options, 69
enabling signal buffer, 99
execution time, 86
execution
 ceasing, 77
 controlling, 77
 go, 77
 optimising speed, 77
 restarting, 85
 step into, 77
 step out, 77
 step over, 77
 step to cursor, 77
external project options
 build mode, 64
 debug build, 64
 processor, 64
 release build, 64
 target name, 64
external projects, 57

F

file defaults, 67
File menu, 46, 47, 105, 157
 open, 67

H

Help, 54
Help menu, 46, 51, 54

I

inactive trace buffer, 85
installation, 5
internal project options, 61
 Assembler, 63
 build, 63
 C compiler, 63
 category, 63
 C-Like compiler, 63
 Linker, 63
 options string, 63
internal projects, 57
interval, 80
interval window, 80

K

keyboard summary, 157

L

list boxes, 52
locals, 112, 114
locating and changing text
 find, 71
locating and finding text
 replace, 72

M

menu operation, 45

O

on-chip peripherals, 102
opening a project, 59
opening files, 67
Options
 Debug, 77
Options menu, 46, 50

debug, 77, 80, 83, 84
editor, 69

P

pin and port windows, 93
pin numbering, 95
plot lines
 disabling, 99
 enabling, 99
 removing, 99
port
 script files, 94, 95
 views, 93, 104
 windows, 93
port simulation techniques, 93
processor information file, 58
program counter, 78, 79
 indicator, 31
project, 34
 batch file, 57, 58
 debug information file, 58
 edit, 60
 external, 61, 64
 internal, 61
 open, 59
 processor information file, 58
 source files, 58
 window configuration file, 58
project files, 57
 external, 57
 internal, 57
Project menu, 46, 48, 74, 157
 execute, 35
pull-up control, 95

Q

quick watch, 111
 add to watch, 112

modify variable, 112
zoom, 112

R

radio buttons, 52
RAM
 customised memory views, 98
 window, 97
re-building the project, 74
registers, 115
roll-back displays, 84
 go back, 84
 step back into, 84
 step back over, 84
 step back to cursor, 84
 step out, 84
run time links, 5, 7

S

script file, 22, 104
 commands, 82
 events, 123
 execution and control flow, 123
script file variables, 120
script files, 23, 35
 breakpoints, 92
 examples of uses, 118
 identities, 127
 port, 94, 95
 purpose and uses, 117
 test panels, 117
script language
 breakpoints, 125
 button events, 125
 comment delimiters, 120
 edit events, 126
 elements of statements, 120
 grammar and syntax, 119

- keywords, 128
- operator precedence and
 - associativity, 122
- operators and expressions, 121
- pin events, 126
- statements and lines, 119
- timeout events, 126
- script language commands and
 - functions
 - ABS, 131
 - ACOS, 131
 - AND, 132
 - ASC, 132
 - ASIN, 132
 - ATAN, 133
 - BREAKPOINT, 133
 - CHR\$, 135
 - CLOSE, 135
 - CONNECT, 135
 - COS, 136
 - DIM, 136
 - EDIT, 136
 - END, 137
 - EQV, 137
 - EVENT, 137
 - EXP, 139
 - FOR-TO-[STEP], 139
 - GETEDIT, 140
 - GETEDIT\$, 140
 - GETFC, 140
 - GETFS, 140
 - GETPC, 141
 - GETTIME, 141
 - GETV, 141
 - GLOBAL, 142
 - GO, 142
 - GOSUB, 142
 - GOTO, 143
 - IF-THEN-[ELSEIF]-[ELSE]-
ENDIF, 143
 - IMP, 144
 - INPUT, 144
 - LEFT\$, 144
 - LEN, 145
 - LET, 145
 - LOCAL, 146
 - MID\$, 146
 - NOT, 147
 - OPEN, 147
 - OR, 148
 - PEEK, 148
 - POKE, 149
 - PRINT, 148
 - REM, 149
 - REPEAT-UNTIL, 150
 - RETURN, 150
 - RIGHT\$, 150
 - SETBITMAP, 151
 - SETEDIT, 151
 - SETFLAG, 152
 - SETPC, 152
 - SETR, 152
 - SETV, 153
 - SGN, 154
 - SIN, 153
 - SQR, 154
 - STOP, 154
 - TAN, 154
 - TIMEOUT, 155
 - XOR, 155
 - setup, 5, 6
 - signal, 24
 - buffer, 99
 - recording box, 98
 - zoom in, 102
 - zoom out, 102
 - signal buffer

- control, 82
- signal plots, 24
- signal recording box
 - axes and scales, 99
 - disabling plot lines, 99
 - enabling buffer, 99
 - enabling plot lines, 99
 - markers and shading, 99
 - pin selection, 99
 - removing plot lines, 99
 - setting up, 98
 - viewing the results, 102
- Signal window
 - control, 82
- source debugging, 111
 - call stack, 111
 - locals, 111
 - quick watch, 111
 - registers, 111
 - watch, 111
- source files, 58
- step
 - multi, 32, 77
 - single, 32, 77, 84
- Step Into, 32, 77, 78
- Step Out, 78
- Step Over, 77, 78
- Step to Cursor, 31, 77, 78
- syntax colouring, 68

T

- Test menu, 46, 49, 50, 105, 157
- test panel, 21, 104
 - button, 106
 - edit box, 108
 - flag, 106
 - move, 106
 - options, 105

- properties, 108
- setting up, 105
- text box, 106
- test panels
 - examples of uses, 118
- trace buffer
 - control, 82
 - inactive, 85
- trace buffering, 37, 83
- Trace menu, 24, 46, 158
 - roll-back displays, 84

U

- user interface, 41
 - desktop, 42
 - icon, 42
 - keyboard, 53
 - menu, 42
 - menu bar, 42
 - menu operation, 45
 - scroll bars, 43
 - status bar, 43
 - tool bar, 42
 - toolbar, 53
 - window, 42
 - window elements, 43

V

- View menu, 46, 48, 157
- viewing simulated objects, 97

W

- watch, 112
 - adding items, 112
- window
 - interval, 80
- window configuration file, 58

window elements, 43

Window menu, 24, 46, 51, 80, 159

device, 97, 104

registers, 115